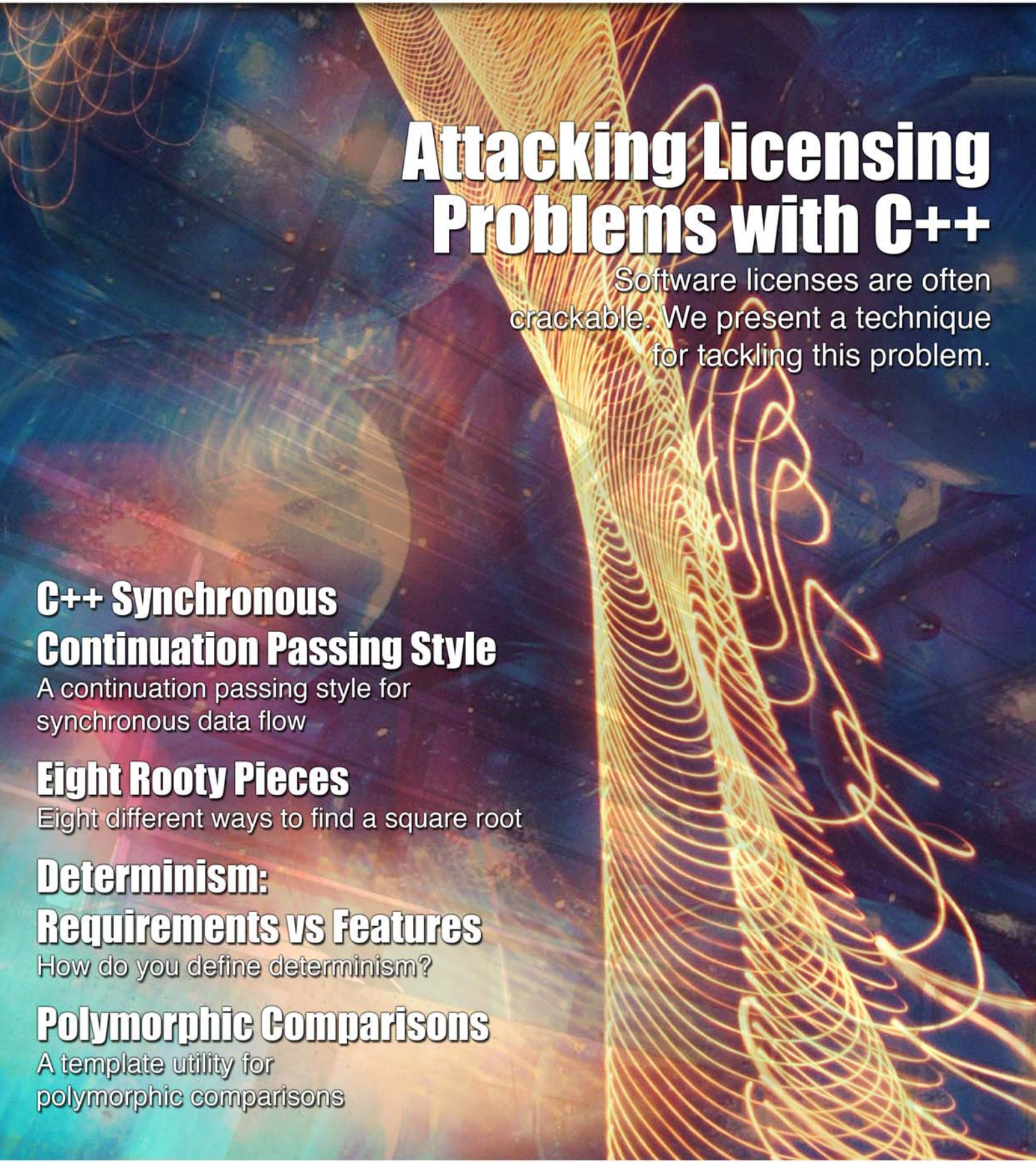
overload 135

OCTOBER 2016 £3



Attacking Licensing Problems with C++

Software licenses are often crackable. We present a technique for tackling this problem.

C++ Synchronous Continuation Passing Style

A continuation passing style for synchronous data flow

Eight Rooty Pieces

Eight different ways to find a square root

Determinism: Requirements vs Features

How do you define determinism?

Polymorphic Comparisons

A template utility for polymorphic comparisons

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 135

October 2016

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 136 should be submitted by 1st November 2016 and those for Overload 137 by 1st January 2017.

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of the ACCU

For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copyrights and Trade Marks

Ain't that the truth?!

Witnesses promise to tell the truth, the whole truth and nothing but the truth. Frances Buontempo wonders what truth actually means.

It is sometimes difficult to tell if two things are identical or equivalent. Other times, it is much easier. If these two pages' worth at the start of *Overload 135* were topical based insights and opinions, it would count as an editorial of sorts. If, instead, it were me trying to unpack what counts as equivalence specifically, or more generally, truth we could be here for a very long time without an editorial. I shall therefore avoid too much philosophy and start with Booleans. Clearly, there is more than one: otherwise we wouldn't say 'Booleans'. George Boole was an English mathematician in the 19th century. He started his career as a school teacher in Yorkshire, but ended up in academia, despite never obtaining a degree. He published many papers, starting with differential equations, but more famously contributing towards the algebra of logic. This algebra operates on two symbols, {0, 1} or an equivalent, combined with AND and OR connectives or operations. Different symbols can be used – the structure of the algebra will not change. Sometimes extra operations, like NOT are introduced. Shannon proved this algebra's use for describing switching circuits, making it "indispensable in the design of computer chips and integrated circuits" [Wolfram].

Some programming languages have a Boolean type, often called **`bool`**, in his memory. One suspects Boolean is too much typing, and shorter names like **`long`**, or **`char`** tend to be preferred. The word **`boole`** may have the same number of letters as **`short`**, but much programming tends to be afflicted by a grand vowel shortage, as often evidenced by variables names or function names. One random selection of code on the internet quickly found a function to draw a line in a cube called **`cubLine`** [RosettaCode]. Mocking code is all too easy though, and to be fair modern IDEs are making this vowel shortage less prevalent, though some languages tend to produce terser code than others and we tend to be stuck with keywords. My personal history of encounters with truth types begins with the very 'shouty' **`BOOLEAN`** in MFC, interspersed with some macros defining **`BOOL`** ending up with **`bool`**. Never work on a code base that has

```
#define TRUE 0
#define FALSE 1
```

I do seriously wonder where the 'e' went though; **`bool`** it is then.

Do we actually require a Boolean type? If we need to perform a set of statements conditionally, we need a way to do a high level equivalent of a jump instruction. Even if we created a fictional language that just had **`JZ`** – jump on zero – the comparison with zero would be made, and the jump performed if the value were zero. This may not require a Boolean type, but is mathematically, or at least philosophically, equivalent to checking the truth of a statement. Some languages do not have any types, and some are quite loose where they do have a type system. It is rather too easy to coerce almost anything to a Boolean in C++. Previously people resorted to the so-called safe bool idiom [Safe Bool], if they remembered, which avoided you being able to compare two totally disparate things which could be treated as bools, such as an **`int`** and a **`std::basic_ios`**. C++11 introduced explicit conversion operators [Stroustrup], providing a neater solution to the problem.

I was amused to find the phrase 'Truthy' used in JavaScript a while ago. The Mozilla Developer Network states that a truthy value is one "that translates to true when evaluated in a Boolean context. All values are truthy unless they are defined as falsy (i.e., except for false, 0, "", null, undefined, and NaN)" [MDN]. Falsy has of course correctly omitted the 'e'. Falsey would be incorrect and silly. Objects are supposed to be True, or is that TRUE, or true rather, when **`ToBoolean`** is called, but **`document.all`** has unique behaviour, or rather a "wilful violation of the ECMAScript standard" [MDN] for legacy code. Aside from this quirk, since all (other) objects are truthy,

```
new Boolean(false)
```

is truthy [Padolsey]. Truthy might be more explicit and honest than C++ accidental coercions, however being able to create a new false object thereby making it true is of note. I shall resist commenting on Variant Bool types with a TRUE value of -1. (Thanks to Chris Oldwood for the reminder.) The truth can be twisted and blurred with great ease in any context. A recent suggestion that a claim in the media was "100% false" just emphasises the fuzziness that happens. What status would a 50% false statement have? Can something really be partially true? First order logic might be clearer; as we move to higher order logic things become less well-behaved [HOL].

Boolean algebra has defining laws – commutativity, associativity and so on – making it an algebra. When combined with other theorems such as De Morgan's laws

!(A&B) = (!A) | (!B) and !(A | B) = !A & !B,

proofs of equivalence between various statements can be made. More generally, a simple truth table allows you to prove equivalence between expressions, thereby simplifying them. Many of us have resorted to symbolic manipulation to neaten up some confusing nested **`if`**s and **`else`**s in code, I'm sure. If the derivation of an equivalent formulation is correct then the code will have identical behaviour; however, when faced with a tangled mess it is safest to have tests to verify this, as we all know. Though Boolean logic works precisely, real code has a tendency to take on a life of its own. As the quote goes "No obvious deficiencies."

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

I have avoided asking what is meant by truth so far. Some may claim mathematics deals with truth, though it is less controversial to say mathematics and logic give us ways to deduce equivalence. If $A \Rightarrow B$ and A is given, B may be concluded. Mathematics also gives us a precise definition of equivalence, though I shall pull back from a maths lesson on posets, cosets and the like. Theorems allow us to draw further conclusions from a given starting point. On many occasions we arrive at a conclusion which may run against our intuition. We may not have discovered a new truth *per se*, but discovering something we believed does not hold true can be startling and exciting in equal measures. That there are more real numbers than whole numbers, even though there are infinitely many of each usually gives people pause for thought. There are many other examples. Our intuition is often incorrect. There are a variety of ways to prove something mathematically. There are a variety of ways that people attempt to prove things; authority, intimidation, tautology, stubbornness, … [Wilson] The toolkit of sound proof is large. We can use proof by contradiction, for example starting with the assumption that $\sqrt{2}$ is rational, we can conclude something inconsistent and are thereby forced, if we are reasonable, to accept that $\sqrt{2}$ is irrational. Othertimes, a simple counterexample will work. All primes are odd. Ah, apart from 2. So not all primes are odd. People do invoke the phrase "The counterexample that proves the rule," though they are missing the point somewhat. I personally love proof by induction, though it can take a while to realise why it works, and several initial attempts accidentally end up assuming that which was to be proved on-route. Using logical equivalence, for example that $A \Rightarrow B$ is identical to $!A \Rightarrow !B$ can inspire a different approaching to proving (or disproving) a given statement, in this case a proof by contraposition. There are many other approaches to proofs. For those interested, some material is available based on an Open University proofs workshop [Stibbe]. We still haven't defined identical of course. Furthermore, do any of these proof methodologies give us truth? I shall leave these questions as an exercise for the reader (proof by boredom?) and stick with the easier claim that such approaches certainly can uncover incorrect intuition and falsify conjectures.

Moving on from Mathematics, the essence of science could be regarded as falsifiability, circumventing the need to define truth or prove anything is true. Karl Popper [Standford] was an eminent philosopher of science. He insisted that a statement or model needed to be falsifiable in order to be scientific. Other types of statements are available but cannot be regarded as scientific. If one observation could falsify a statement, such as all swans are white, this is a genuine theory. On the other hand, he held that Freud's psychoanalytic 'theories' were unfalsifiable stories, and had similar views on Marx's account of history. Neither is science. Though both may seem to provide a model that fits observations, there is no way to prove them incorrect so they must remain as fiction rather than science. When we debug a chewy problem we often have a spark of intuition which we weave into a story to explain the observed behaviour. We must then try specific observations to ascertain whether our tale is in fact correct, so our tale must also relate to things we can observe. The alternative might be just hitting things with a hammer until they work. This is a tongue in cheek way to disambiguate science and engineering though. If our code has worked fine up to now, this is a falsifiable statement. We can keep observing and see if this continues. We can even try to make it break, say under load. We can never conclude our code is verifiably correct. "Conclusively falsifiable is not conclusively verifiable." [Standford]

Some branches of computing use proofs of correctness, though these appear to be quite niche. In fact, formal verification strictly speaking would require a proof of termination which takes us to the halting problem. This can be avoided by proving partial correctness – that if an answer is returned, it will be correct. Does this mean all program of the form

```
while (true) ;
```

are partially correct? It is often joked that mathematics tends to be exactly and precisely correct but not much use. There are many similar jokes, but the physicist and engineer lost in a hot air balloon over a field, asking a mathematician where they are, are fabled to be given the correct, but useless answer, "In a hot air balloon." Precisions and proofs are useful, in the right context. There is more to life than a stick utilitarian stance though. Some things are beautiful, or surprising, or just fun. Some things end up being useful at a much later date. Complex numbers might initially seem like a very abstract concept, but they can make the mathematics of electronic circuits easier. This is not why they were introduced. Starting with the observation that the square of any natural number is a natural number, and the square of any integer is the natural numbers plus zero, means negative numbers have no square root. Suppose they do. Call $\sqrt{(-1)} = i$ and see what happens. You could ask why i, what's wrong with j, or even k. You could use all three, and let $ijk = -1$, giving what's known as the Quaternions, arriving at non-commutative numbers. Asking 'Why?', or 'What if?', can end up at some surprising and counter-intuitive places.

In order to prove something mathematical you often need to start with one or two specific examples to form some intuition before proceeding more formally. Indeed, that can provide a counterexample. Intuition, though often incorrect, can be useful. With practise in a given realm, you can sharpen your intuition. Mechanics can often diagnose a potential cause of a problem, by listening, or even smell. When our code goes wrong, we do sometimes have a gut feeling about the area to look in or the sort of problem to go hunting for. We need to avoid using one hunch as a tool to apply to everything else in sight though. If we've been stung by an off-by-one error, we can then tend to assume this is the cause of anything else odd we see. When you have a hammer, everything looks like a nail. Our circle of influence can limit our approaches too. If everyone around us insists on unit testing, we will be horrified if we end up meeting people who don't unit test. If we follow a group of like-minded people on Twitter, and thought Brexit was a terrible idea, we'd be taken aback when the referendum voted for Brexit. We do end up surrounding ourselves with people we tend to agree with, listening in echo chambers. We can also end up searching out references that back up our position. Confirmation bias creeps in to many areas. People seek out positive data, and disregard negative data. 'See I told you so,' when one example fitting a theory presents itself, but never an "Oh, perhaps I was incorrect" when falsifying data rears its head. We should avoid echo chambers, be aware of our assumptions, and realise we aren't always right. Like-minded people can spark creativity though, and sometimes you need some basic assumptions to even get things going. Intuition can be the starting point of ideas too. History (or Gödel) has shown the strict logic cannot ever give a complete and consistent framework. Einstein said,

> There is no logical path leading to these ... laws. They can only be reached by intuition, based upon something like an intellectual love of the objects of experience. [Stanford]

Do what you love, ain't that the truth.

References

[HOL] https://en.wikipedia.org/wiki/Higher-order_logic

[MDN] https://developer.mozilla.org/en-US/docs/Glossary/Truthy

[Padolsey] http://james.padolsey.com/javascript/truthy-falsey/

[RosettaCode] http://rosettacode.org/wiki/Draw_a_cuboid#Perl

[Safe Bool] https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Safe_bool

[Standford] http://plato.stanford.edu/entries/popper/

[Stibbe] http://www.shirleenstibbe.co.uk/proofs-2/4557195004

[Stroustrup] http://www.stroustrup.com/C++11FAQ.html#explicit-convertion

[Wilson] http://jwilson.coe.uga.edu/emt668/emat6680.f99/challen/proof/proof.html

[Wolfram] http://mathworld.wolfram.com/BooleanAlgebra.html

Determinism: Requirements vs Features

A program can easily be non-deterministic. Sergey Ignatchenko considers how to define determinism.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

As was discussed in a blog post [NoBugs16] a few months ago, determinism can have quite a few important practical uses, ranging from replay-based regression testing, to low-latency determinism-based fault tolerance, with production post-mortem in between.

In the very same post (as well as in *Overload* [NoBugs15a]) requirements to achieve determinism were discussed; however, one point was left out of the deliberations, and this is the question of '*what exactly is the definition of determinism* our system needs to comply with, to achieve the deterministic goodies mentioned above'. This article aims to provide some analysis in this regard.

First of all, let's mention that, in practice, at least three different types of somewhat deterministic behavior can be observed; the differences between them are related to changes which can break deterministic behavior.

Types of determinism

Cross-platform determinism – an extremely difficult one

The most obvious form of determinism (and usually the one which comes to mind when speaking about determinism without specifying further details), is what I call *cross-platform determinism*. A program which is cross-platform deterministic has the following properties.

Definition 1. A program in source code form is considered to be *cross-platform deterministic* if and only if:

- When the source code of the program is compiled by several different compilers across several different platforms, the resulting executable produces exactly the same results given exactly the same inputs.
- For those platforms where it cannot produce exactly the same results, ideally such a program shouldn't compile at all (or at least should fail immediately after being started).

Notes:

- This should stand for *all* acceptable inputs
  - Ideally, non-acceptable inputs should be filtered out by the program (for example, asserted or ignored).
- If the program is interactive (i.e. it interacts with the world outside itself), *all* the interactions with the outside world need to be considered as program inputs.
  - This also applies to non-deterministic system calls such as 'current time'; see the discussion on ways to implement this in 'Deterministic Components for Distributed Systems'[NoBugs15a].

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

Factors breaking cross-platform determinism

Cross-platform determinism is the strictest definition of determinism I know; not surprisingly, there are quite a few factors which can break it:

1. **CPU compatibility issues**. Just as one example – if the CPU has non-IEEE-compliant floating-point arithmetic, it can easily break cross-platform determinism. The same goes for CPUs with bugs (such as an infamous Pentium FDIV bug). NB: even IEEE-compliant floating point *per se* doesn't guarantee determinism; see 'Compiler compatibility issues'.
2. **Compiler compatibility issues**. It just so happens that compilers can generate code which produces subtly different results depending on the platform. In particular, some compilers are known to rearrange floating-point calculations – which is not exactly correct (as floating-point addition is non-associative due to non-linear rounding); another example of problems relate to 'what does the compiler use for intermediaries' [RandomASCII]. These issues are also known to depend heavily on compiler settings ☹.
3. **Runtime library compatibility issues**. Even standard libraries leave quite a bit of leeway to implementers (at least in C/C++). Just as one example – if we have a partially ordered collection (such as `multimap<>`), then iteration over this collection doesn't specify a 'correct' order for those items with equal keys; as a result, two perfectly compliant implementations can produce rather different results, breaking cross-platform determinism as specified above. Floating-point libraries are known to introduce quite a bit of not-exactly-matching behavior too.
4. **C/C++: Reading dirty RAM, and other 'Undefined Behavior' stuff**.
5. **C/C++: Using pointers for anything except for dereferencing**. Especially dreadful in the presence of ASLR (Address Space Layout Randomization), but has been seen to cause severe problems in other cases too.
6. **Multithreaded stuff**. As a rule of thumb, multithreaded programs as such are *not* deterministic. They *can* be made deterministic by restricting the multithreaded model to certain limited patterns of inter-thread interactions.
   a. My (by far) favorite example of a deterministic multithreaded program is having SHARED-NOTHING REACTORS as described in [NoBugs15a] / [NoBugs16], with all the inputs of each REACTOR separately considered as program inputs. This way, we make each individual SHARED-NOTHING REACTOR deterministic, effectively removing multithreading from scope.

With such a long list of potential troubles, it is no wonder that achieving cross-platform determinism is extremely difficult

b. SHARED-NOTHING REACTOR is not the only possible way to ensure determinism. Strictly speaking, even mutex-based inter-thread synchronization can be made deterministic; however, to do it, we'll need to consider the whole state of the object protected by mutex to be program input at this point, which will reduce the practical uses of this approach to a pretty much empty set.

With such a long list of potential troubles, it is no wonder that achieving cross-platform determinism is extremely difficult (at least for C/C++). In practice, it has been observed that it is items #2 (compiler compatibility) and #3 (runtime library compatibility) which tend to cause the most problems. Item #1 is usually not *that* bad (though YMMV), and items #4–6 are in our hands, so we can avoid them.

Which leads us to the following observation (which is well-known in gamedev circles):

> Achieving cross-platform determinism for a sizeable program ranges from 'extremely difficult' to 'next to impossible' ☹

However, taking a look at the list above (and our notes about things which tend to cause the most trouble), we can try to limit our deterministic appetites to the very same platform – and even to the very same executable.

Same-executable determinism – the easiest one

Let's change our Definition 1 to the following

Definition 2. A program in source code form is considered to be *same-executable deterministic* if and only if:

- When the source code of the program is compiled on a single compiler for a single platform, using the same libraries, the resulting executable produces exactly the same results given exactly the same inputs.

Note: the same notes as for Definition 1 still apply.

As follows from Definition 2, the same-executable deterministic program no longer suffers from breaking-determinism factors #1 (well, save for an occasional FDIV bug), #2, and #3. This makes it *much* more realistic for being implemented in practice (and yes, it has been done more than once too).

Same-platform determinism against minor changes – in-between one

To implement some features (mostly this applies to Regression Replay Testing), a same-executable determinism is not sufficient; what we need is something along the lines of the following Definition 3:

Definition 3. A program in source code form is considered to be *same-platform deterministic against minor changes* if and only if:

- It is *same-executable deterministic*, ***and***
- When relatively small changes to the source code are made (creating 'new' source from the 'old' one), and these changes break determinism in an unmodified piece of code, the number of changes to the source code which are necessary to restore determinism (so that the 'new' executable produced with the same platform + compiler + libraries but produced out from the 'new' code, behaves exactly as the 'old' one with regards to unmodified portions of the code), is relatively small too.

Note: same notes as for Definition 1 still apply.

The second condition in Definition 3 is necessary to deal with scenarios when minor changes to the code break determinism (for example, it may happen because of the compiler using a different reordering of floating-point operations for different executables); however, such occurrences of non-determinism should be identifiable and locally fixable.

Of course, any definition which says something is minor is inherently vague, and yet in practice I've seen these kind of things working reasonably well. Usually, it goes along the following lines:

- the code is maintained as *almost* cross-platform deterministic. More specifically, it is written with the intent to be 100% cross-platform deterministic – and as soon as any non-determinism is spotted, it is fixed. This is not that difficult; the real difficulty lies in getting from *almost* cross-platform determinism to *real* cross-platform determinism (and the main obstacle to this approach is that spotting rarely occurring non-determinism is difficult, especially when it comes to floating-point stuff – because it doesn't manifest itself often).
- when we have a need to exploit this type of determinism, we're always working with 'old' source code and 'new' source code. And if non-determinism is spotted in 'new' source – it can (and should) be fixed, just as any with other kind of non-determinism. More on this in the 'Replay-based regression testing' section below.

One really simple example to illustrate this might go as follows. In our 'old' source code, we have something like

```
double f(float a, float b, float c) {
  //do something
  return a + b + c; //(1)
}
```

Usually, the formula is much more complicated than that, but this one will do for our purposes. In fact, the line is highly likely to be non-deterministic but we didn't spot it (or didn't care at that point). And let's assume (just for the sake of defining things more precisely) that the compiler interpreted it as

```
double f(float a, float b, float c) {
  //do something
  double tmp = (double)b + (double)c;  //(2)
  return (double)a + tmp;              //(2)
}
```

Note that while this is a perfectly valid interpretation of our first sample, it is not *the only* valid one. For example, a compiler might add `b` and `c` as floats, and only then convert it to a `double`, or it might use a different order of additions. Any such variation would produce *almost* the same – but not identical – results.

a compiler can rearrange things to use a different kinds of intermediaries, or a different order of floating-point additions

As a result, when we change some code *near* line (1) – for example, the 'do something' part, a compiler can rearrange things to use a different kinds of intermediaries (because it has different registers available), or a different order of floating-point additions (just because it felt that it would allow for better use of a pipeline for this specific target CPU). As a result, our new code can start to behave differently from the old one. As the difference is about extreme corner cases, it may or may not pop up during our testing. However, from the point of view of our Definition 3 (and in particular, from the point of view of replay-based regression testing as discussed below) we're fine in both cases:

- if the difference didn't manifest itself during testing, then for the purposes of these specific tests, our code is still perfectly deterministic (!). In other words, as long as we cannot observe that the program is non-deterministic, in the context of specific input vectors we don't care about it.
- if the difference did manifest itself during the testing, it can be identified, and the line (1) can be rewritten into two lines (2), making the 'new' code deterministic (and consistent with the 'old' code too). Strictly speaking, this second property (consistency with the old code) is not guaranteed; however, most of the time finding a deterministic version of the new code which is equivalent to the old one is perfectly feasible.

Deterministic goodies

Now, let's list those goodies which we can get out of determinism – and see which type of determinism is required for each one.

Deterministic lockstep etc.

Description. One common example of a reason to use determinism (in particular, in games) is to produce exactly the same results across different computers. In this case, it would be possible just to send the same inputs across the network to all the computers (and for games, the inputs are usually very small) and to get all of the computers to run exactly in sync. One notable example of such a protocols is *deterministic lockstep* [GafferOnGames].

Required Determinism. To make deterministic lockstep (and other similar protocols) work across clients running on different platforms, we need *cross-platform determinism* as defined in Definition 1 ☹. Unfortunately, it is rarely possible (and to the best of my knowledge, most such attempts have failed ☹).

Client-side replay

Description. Another common example of determinism-based features (also coming from the gamedev world) is client-side replay. In such cases, we record only the inputs of the game, and then replay it by simply feeding the same inputs to the client.

Required Determinism. To make client-side replay work across clients running on different platforms, we also need *cross-platform determinism* as defined in Definition 1 ☹.

Production post-mortem

Description. As described in [NoBugs15a], if we have deterministic REACTOR, then we can write a log of all the events for that REACTOR. Then, if something bad happens (like a crash or an assert failure), we have not only the current state, but *the whole history* of the events which led to the crash. We can replay this history in the comfort of a developer's machine to reproduce the bug 100% of the time because of the behavior being deterministic (and a reproducible bug is pretty much a dead bug).

In practice, when saving the whole history is not practical (and it usually isn't ;-)), we can still have a circular buffer storing the *last N seconds of the program before the crash*. While this doesn't allow identification of *all* the bugs out there (because the bug condition could have occurred before those *N* seconds), for quite a few systems it still allows identification of 80–90% of them.

Required Determinism. To make production post-mortem work, only *same-executable determinism* (as defined in Definition 2) is necessary (well, usually it is not a problem to store all the released executables).

Low-latency fault tolerance

Description. As described in [NoBugs15b], deterministic REACTORs (with circular logging) can be used to achieve low-latency fault tolerance (in a sense, it is ideologically similar to the now-discontinued 'Virtual Lockstep' technique which was used by VMWare). Such determinism-based implementation of fault tolerance allows latencies which are inherently better than those of 'Fast Checkpoints'.

Required Determinism. For determinism-based fault tolerance to work, we only need *same-executable determinism* (as defined in Definition 2). That's because after the catastrophic server failure, we'll use exactly the same executable to achieve exactly the same results.

Replay-based regression testing

Description. As it was described in [NoBugs16], the same REACTORs with input logging can allow the use of real-world inputs to test that certain changes didn't really change the behavior of the system. While such testing is inherently limited to the testing of (a) refactoring and (b) new features (and is not applicable to the testing of changes) – it can still facilitate testing quite a few things in an extremely reliable manner (and it is especially important as most of development is about new features).

The idea for such testing goes along the following lines:

- record all the program inputs while the old code runs in production (usually this is done on per-REACTOR basis)
- make changes, producing new code (and a new executable)
- run a replay of the recorded inputs against the new executable, and compare the results with those of the old code. Any changes indicate that 100% regression is not achieved.

Required Determinism. To get the benefits from replay-based regression testing, we need to have *same-platform determinism against minor changes* as defined in Definition 3.

In practice, this is often possible. While small changes *can* cause different behavior (in particular, with floating-point order and intermediaries) – it is usually not that difficult to fix them (in the case of floating-point issues due to compiler optimizations, by removing ambiguities and enforcing the behavior which was used by the old code, see example above). As soon as the regression test passes, this floating-point disambiguation can be rolled back if desirable; this can be done as a separate stage, and although it will be breaking strict regression testing, with the change being trivial, it can be reviewed for near-equivalence very easily.

Features-vs-determinism-type matrix

Now, we're in position to summarize our findings in the following table:

	Same-Executable Determinism (Definition 2) – the simplest	Same-Platform Determinism against Minor Changes (Definition 3)	Cross-Platform Determinism (Definition 1) – most complicated
Deterministic lockstep			Yes
Client-side replay			Yes
Replay-based regression testing		Yes	Yes
Production post-mortem	Yes	Yes	Yes
Low-latency fault tolerance	Yes	Yes	Yes

Conclusions

We've analysed different types of determinism (as encountered in the real world), and figured out which of these types of determinism are required to obtain different benefits.

From a practical point of view, this means that while deterministic lockstep and client-side replay are not usually feasible if multiple platforms are involved, goodies such as replay-based regression testing, production post-mortem, and low-latency fault tolerance are usually well within reach. ■

References

[GafferOnGames] Glenn Fiedler, Deterministic Lockstep, http://gafferongames.com/networked-physics/deterministic-lockstep/

[Loganberry04] David 'Loganberry', 'Frithaes! – an Introduction to Colloquial Lapine!', http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs15a] 'No Bugs' Hare, 'Deterministic Components for Distributed Systems', *Overload* #133 (June 2016)

[NoBugs15b] 'No Bugs' Hare, 'Server-Side MMO Architecture. Naïve, Web-Based, and Classical Deployment Architectures', http://ithare.com/chapter-via-server-side-mmo-architecture-naive-and-classical-deployment-architectures/

[NoBugs16] 'No Bugs' Hare, 'Modular Architecture: Client-Side. On Debugging Distributed Systems, Deterministic Logic, and Finite State Machines', http://ithare.com/chapter-vc-modular-architecture-client-side-on-debugging-distributed-systems-deterministic-logic-and-finite-state-machines/

[RandomASCII] Bruce Dawson, 'Floating-Point Determinism', https://randomascii.wordpress.com/2013/07/16/floating-point-determinism/

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

Eight Rooty Pieces

Finding a square root is a common interview question. Patrick Martin demonstrates eight different ways to find a root.

Sigh Some things we have to deal with...like interview questions. Recently I've been interviewing candidates a bit more and naturally some old coding exercises I've collected over time have come to the fore, along with some impressions I've developed.

Let's assume it's that time in the interview when the candidate shows signs of being suitable to step up to the next level. At this point it really starts to matter whether the interviewer has prepared sufficiently well for this eventuality. Therefore, a question that has several such plateaus to provide some good challenge for the candidates who are on a roll would be very useful. I'm also suggesting the topic should generate discussion points so that in the initial 15 minutes that the candidate and I are forming a mutual opinion, I will get (and generate) as representative an impression as possible. Remember, the candidate is also interviewing you, and they might well form an opinion if all you're asking them to do is regurgitate facts.

So are there interview questions that have genuine 'breadth and depth'?[1]

Well, here's a fun little question I've been carting along to interviews in note form for some time that I aim to persuade you will generate discussion points, and my notes have grown to either being

- a significant number of sheets of paper
- or one page of an entirely unusable font size

So, without further ado.

The question

> Please implement the square root function
> [Wikipedia_1] [monkeys_sqrt]

One thing I like about this question as that it's really quite easy to run and test even in some minimal web based online coding tool.

What one learns in asking this question

- First up: some people are really quite wary of `sqrt()` in this context. I am not judging, let us be clear.
- There is a giant range in the comfort level for working through the issues in implementing this deceptively simple function.
- People are generally wrong to be frightened of the problem.

  They often surprise themselves when they reach the end.
- There are quite a few approaches that are recognisable.

5.000000 stages of shock.

It would be a fair point that there is a sneaky element of testing character and resilience with this question. I am going to argue this is both legitimate and worthwhile, based on my assertion that [i] it's not that hard and [ii] there is so much to discuss that running out of steam / time is not that much of an issue in the wider scheme of things.

Nevertheless it seems people pass through shock and a number of other stages when presented with this challenge: Denial, Anger, Bargaining, Depression. I would like to think we can short-circuit this and skip straight to Acceptance (and perhaps a little Fun?). Let's dive in and see what I'm talking about.

Initial unstructured points

The exercise typically goes through a number of phases, sometimes the first of which is akin to scoping out the problem.

This can be a very revealing phase: demonstrating the candidate's process for collecting information. Amusingly, some make adequate assumptions and plough on, because as we will see later: 'double is just fine'[2], whereas some might ask about which arbitrary precision packages we're allowed to use.

Assuming we're here though: here's an incomplete list of things one might want to touch upon

- what is the return type?

  discussion points might be considering arbitrary precision
- what's the input type?

  discussion points – is it the same as the return type, what bit size is the range, compared to the domain?[2]
- what happens for inputs of 1, > 1, < 1 or negative values?

  is this going to influence your thinking on the approach you take?
- what is your criterion for accuracy?
- how about float denormal values inputs, results [Wikipedia_2]
- what about NAN, NaNQ, NaNS? [Wikipedia_3]
- 'Oh hey, what do CPUs do?' *discussion points*[3]

  you may want to keep your powder dry when asked, so push it, and pop it later
- finally, $bright_spark may well know the POSIX prototypes [posix].

  These prototypes address a lot of the above questions

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

1. Why are we using questions?
2. For IEEE 754 double, the maximum sqrt will exceed the maximum value for IEEE 754 float, so this forces us to consider the same return type as the input type.
3. These might be using dedicated FPU hardware or native CPU commands. In the silicon itself, one might find GoldSchmidt's method, or Newton Raphson; *Some Assembly Required* [SAR] has a large number of interesting comparisons, including old and modern native SQRT instructions.

Patrick Martin Patrick's github repo was classified using a machine learning gadget as belonging to a 'noble corporate toiler'. He can't top that. Patrick can be contacted at patrickmmartin@gmail.com.

the name of the game here is to get discussion points, any and all means are acceptable

Eight approaches

So, having got past the initial stage of get to know the question, it's probably time to start writing code. Here follow eight implementations of varying quality, nominally in C++.

Caveat

Please remember that for some of these implementations, it may be hard to find canonical examples 'out there' of some of these algorithms. This is because they are in fact a bit rubbish. The more 'recognisable versions' are pretty much shadows of the many already thoroughly written-up versions available for research. Remember though, the name of the game here is to get *discussion points*, any and all means are acceptable.

Alien technology

An additional benefit of these discussions is when a novel-looking implementation arises, having some preparation under your belt will serve you well in recognising a variant of one of the following principles and steering the code/conversation in a more productive direction for *discussion points*.

'One liners'

Closed form FOR THE WIN

Explanation: closed form for the win!

```
return exp(0.5 * log(val));
```

This hinges on the identity

$$\log x^y = y \log x$$

and if we remind ourselves that the power that generates a square root is 0.5, and exp is the inverse of log

$$\text{sqrt}(x) == x^{1/2}, \log(\exp(x)) == x$$

it all drops into place.[4]

Note that I did eliminate `pow(x, 0.5)` as a possible solution as that felt a bit too much like cheating to me.

Search algorithms

This class of solution hinges on iterating upon a trial value until convergence is attained – I've introduced a `seed_root()` function with no explanation that returns a 'good initial guess' for sqrt() in order to concentrate on the details. We'll come back to `seed_root()` later on.

The Babylonian method or Hero's method

The graphical explanation of this algorithm is: iterative search for square root by successive reduction of difference in length between the 2 sides of a rectangle with the area of the input value. [Wikipedia_4]:

```
double my_sqrt_bablyonian(double val) {
  double x = seed_root();
  while (fabs((x * x) - val) > (val * TOLERANCE))
  {
    x = 0.5 * (x + (val / x));
  }
  return x;
}
```

Listing 1

pick side
derive other_side by A / side
if side == other_side: return side
else split the difference for the next side and loop

and hence Listing 1.

The loop is controlled by a test on whether we're 'near enough' to the answer, which may be a *discussion point*. Also note the mechanism for generating a new trial value always narrows the difference between the trial and trial / input.

Notable points:

- it's quite possibly the only algorithm to be presented here that you can implement using a piece of rope and a setsquare. See [Wikipedia_5] for the classical Ancient toolset
- this algorithm is somewhat unique in that it can handle finding the negative root if the trial value passed in is negative
- there is one more interesting fact we will discover shortly

Although there is the amazing Bablyonian Tablet YBC 7289 [YBC7289], it's hard to find a lo-fi image of this implementation so I persuaded a 12-year old to do it for me. Figure 1 shows a Hero's Method contemporary reimplementation for the value 23. We started with a trial value of 6 and got the result 4.8 which is accurate to 0.08%.

Note the Babylonian tablet has sqrt(2) to 9 decimal digits of precision – how did they do that?

Finding the root using Newton Raphson

Explanation: Newton Raphson [Wikipedia_6] searches for the value of x yielding zero for x^2 - value, (hence x^2 = value)

Graphical explanation:

pick a trial value
search for the zero
by building the line passing through
the current trial output with the gradient
of the function at that point
– a numerically estimated gradient will do, for *discussion points*.
the intersection of that triangle with zero is the new trial
exit when desired accuracy attained

Listing 2 is one interpretation.

4. When multiplied, powers are added, hence sqrt is pow(0.5). Two very good examples of working through this identity are available at [SO_1].

Having encountered the two methods independently, I missed the equivalence between them until I took a look at the iteration values

```
double my_sqrt_newtonraphson(double val) {
  double x = seed_root();
  while (fabs((x * x) - val) > (val * TOLERANCE))
  {
    // x * x - value is the root sought
    double gradient =
      (((x * 1.5) * (x * 1.5)) -
       ((x * 0.5) * (x * 0.5))) / (x);
    x = x - ((x * x - value) / gradient);
  }
  return x;
}
```

Listing 2

For *discussion points* see also the related Householder methods [Wikipedia_7]

Newton Raphson with a closed form identity for the gradient

Now, some may know that there is a very simple result $d(x^2)/dx = 2x$ for the gradient that is needed for Newton Raphson and hence plugging in the closed form result for dy/dx, we can skip some typing to yield this (see Listing 3).

Note the original expression containing the gradient:

double gradient = (((*x* * 1.5) * (*x* * 1.5)) - ((*x* * 0.5) * (*x* * 0.5)));

This is the lazy man's version of calculating the gradient around the domain value *x* using the values at *x* +/- *b*.

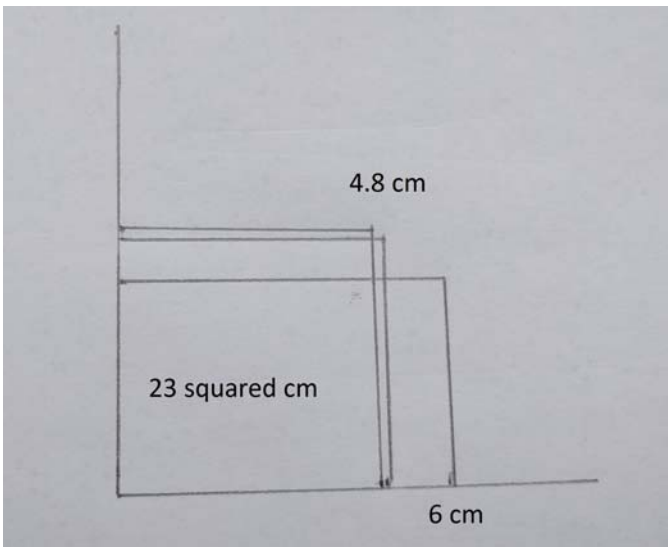


Figure 1

```
double my_sqrt_newtonraphson(double val) {
  double x = seed_root();
  while (fabs((x * x) - val) > (val * TOLERANCE))
  {
    // x * x - val is root sought
    x = x - ((x * x - val) / (2 * x));
  }
  return x;
}
```

Listing 3

$$(x + b)^2 - (x - b)^2 / 2b$$
$$= x^2 + 2bx + b^2 - x^2 + 2bx - b^2 / 2b$$
$$= 2x$$

If *b* were a constant this would not scale with the value of *x*; however, *b* can be substituted by *x*/2 and we recover the initial gradient calculation, and hence an equivalent expression for the closed form expression.

Confession time: I first picked 0.5 * *x* and 1.5 * *x* intuitively, having been hand-bodging numerical estimates into code for some time now, so I didn't think too hard about it (this time around) and serendipitously hit a solution that can be transformed using simple algebra into the closed form solution.

3.0, 2.0 or 1.0 methods?

So far the last 3 solutions have used identical outer loops, merely with different expressions for generating new trial values in the middle. Let's take a closer look at that expression: with the closed form for the gradient we get this expression:

$$x = x - ((x * x - \text{value}) / (2 * x));$$
$$\Rightarrow x = 0.5 * (2x - (x - (\text{value} / x)))$$
$$\Rightarrow x = 0.5 * (x + (\text{value} / x))$$

This is the Hero's method expression, so the final notable point about Hero's method is that it's a condensed version of the more taxing Newton Raphson approach.

Confession time

Having encountered the two methods (Babylonian and Newton Raphson) independently, I missed the equivalence between them until I took a look at the iteration values.

Another confession – even with the mathematical equivalence, there was still a difference as the version just shown has an issue: it fails to locate values for roots above sqrt(`std::numeric_limits::max()`). This is due to an overflow in the expression to generate the new trial value.

The fix – perhaps unsurprisingly enough – is thus:

- double *x* = seed_root();

+ **long** double *x* = seed_root();

If this is found in the wild it would probably be best to put it out of its misery

```
double my_sqrt_range(double val) {
  double upper = seed_root(value) * 10;
  double lower = seed_root(value) / 10;

  double x = (lower + upper) / 2;
  int n = 1;

  while ((n < RANGE_ITERATIONS) &&
    (fabs((x * x) - value) > (value * TOLERANCE)))
  {
    if (((x * x) > value))
      upper = x;
    else
      lower = x;
    x = (lower + upper) / 2;
    n++;
  }
  return x;
}
```

Listing 4

Another set of *discussion points* arise from the necessity of introducing the long version of the type in the algorithm. Is this choice leading to an implicit conversion in the return statement a maintenance wart? What if we need this to be a generic algorithm, parameterised on the input type?

Slow but sure (?)

A range reduction approach

Graphical explanation: a range reduction approach which aims to halve the range [upper, lower] upon each iteration (does not rely upon a particularly good initial guess, though the bounds do need to be ordered). Newton Raphson / Hero can be proven to converge quadratically [Wikipedia_8], whereas this approach effectively converges linearly, hence it requires many more iterations. The algorithm takes 30 iterations for a double sqrt as achieving over 10 digits of decimal precision will typically require approximately 30 halvings of the interval. (See Listing 4.)

If this is found in the wild it would probably be best to put it out of its misery. The possible benefit of this is that candidates less confident of their mathematics will be able to implement this by concentrating purely upon the logic of searching.

Scan and step reduction

This is a very naive guess step and scan approach, reversing and decreasing the step on each transition from above to below. Feed it a decent enough initial guess and it will work its way towards the solution, as it is another linearly convergent solution. (See Listing 5).

```
double my_sqrt_naive(double val) {
  int n = 1;
  double x = seed_root(value) / 2;
  double step = x / 4;
  double lastdiff = 0;
  double diff = (x * x) - value;

  while ((n < RANGE_ITERATIONS) &&
         (fabs(diff) > (value * TOLERANCE))) {
    if (diff > 0)
      x -= step;
    else
      x += step;

    if ((diff > 0) != (lastdiff > 0)) {
      step = step * 0.5;
    }
    lastdiff = diff;
    diff = (x * x) - value;
  }

  return x;
}
```

Listing 5

'Homage to Carmack' method

Finally, the origin of seed_root() can be revealed. Yes, just for fun, an old example of a very fast approximate inverse square root. Here is the obligatory xkcd reference [xkcd_1]. This still works (on Intel), and there is also a good write-up of how this works [Wikipedia_9]. Note there are other values for the magic value than 0x5f375a86 – which oddly get more search hits in Google(?!!).

The original code, sadly has comments and `#ifdef` rendering it unsuitable for printing in a family oriented programming publication, so Listing 6 is a modified version from Stack Overflow [SO_2], and Listing 7 is a version supporting `double`, with the appropriate 64-bit magic value.

The result is not super accurate, but works in constant time and can be used as a seed into another algorithm.

For the most condensed explanation as to how that even works, see the closed form solution and consider that the bits of a floating point number when interpreted as an integer can be used to approximate its logarithm.

'Also ran'

In the grand tradition of sort algorithms [Wikipedia_10], one could always break the ice by discussing solutions that make brute force look cunning.

```
float my_sqrt_homage_to_carmack(float x) {
  // PMM: adapted from the doubly cleaner
  // Chris Lomont version

  float xhalf = 0.5f * x;
  int i = *(int *)&x;
    // get bits for floating value
  i = 0x5f375a86 - (i >> 1);
      // gives initial guess y0
  x = *(float *)&i;  // convert bits back to float

  // PMM: initial guess: to within 10% already!
  x = x * (1.5f - xhalf * x * x);
    // Newton step, repeating increases accuracy

  return 1 / x;
}
```

Listing 6

brutesqrt

```
d = min_double()
while true:
  if (d * d == input) return d
  d = next_double(d)
```

bogosqrt (homage to bogosort)

```
d = random_double()
while true:
  if (d * d == input) return d
  d = random_double()
```

This and the prior approach will need an approach to define the accuracy of match. And perhaps a rather forgiving user calling that code.

Quantum computer method

```
for value in all_doubles:
  return value if value ^ 2 == input
```

It would be hoped that parallelising this would lead to good wall clock times?

Code and tests

Code demonstrating C++ implementations with tests of all the following are available at:
http://www.github.com/patrickmmartin/
2.828427124746190097603377448419 4

Conclusion

So, let's review what we can get out of 'implement sqrt()' in terms of *discussion topics*: closed form results versus algorithmic solutions – discussion on the many interesting properties of floating point calculations, bronze age mathematical algorithms, consideration of domains and ranges. I haven't even touched upon error handling, but it's needed.

And finally there are other really fascinating techniques I haven't touched upon as I judged them too abstruse for an interview scenario: like Lagrange's continued fractions [Wikipedia_11], and also the Vedic techniques mentioned in [Wikipedia_1].

You may have some questions.

Here's my attempt to anticipate them.

1. What's with the name for the repo?

   It's the square root of 8, the number of methods, of course cube root would be have yielded a simpler name – presaging the next installment! Of course, there will be no next installment, as one thing we have learned is that this topic is a giant nerd trap [xkcd_2]. Merely perusing the references to this article for a short time will show how many areas of exploration exist to be followed.

2. Will the Fast sqrt work on big-endian?

   Very funny. ■

```
double my_sqrt_homage_to_carmack64(double x) {
  double xhalf = x * 0.5;
  // get bits for floating value
  long long i = *(long long *)&x;
  // gives initial guess y0
  i = 0x5fe6eb50c7b537a9 - (i >> 1);
  // convert bits back into double
  x = *(double *)&i;

  // one Newton Raphson step
  x = x * (1.5f - xhalf * x * x);

  return 1 / x;
}
```

Listing 7

Acknowledgements

I would like to take the opportunity to thank Frances Buontempo and the *Overload* review team for their careful review comments.

Gabriel Martin recreated the ancient world glories of calculating the square root of 23.

Also thanks to Hillel Y. Sims for spotting an issue in a code sample that got past everyone.

References

[monkeys_sqrt] http://www.azillionmonkeys.com/qed/sqroot.html

[posix] http://pubs.opengroup.org/onlinepubs/9699919799/functions/sqrt.html

[SO_1] http://math.stackexchange.com/questions/537383/why-is-x-frac12-the-same-as-sqrt-x
although the alleged duplicate has a beautiful answer:
http://math.stackexchange.com/questions/656198/why-the-square-root-of-x-equals-x-to-the-one-half-power

[SO_2] http://stackoverflow.com/questions/1349542/john-carmacks-unusual-fast-inverse-square-root-quake-iii

[SAR] http://assemblyrequired.crashworks.org/timing-square-root/

[Wikipedia_1] https://en.wikipedia.org/wiki/Methods_of_computing_square_roots

[Wikipedia_2] https://en.wikipedia.org/wiki/Denormal_number

[Wikipedia_3] https://en.wikipedia.org/wiki/NaN

[Wikipedia_4] https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method

[Wikipedia_5] https://en.wikipedia.org/wiki/Compass-and-straightedge_construction

[Wikipedia_6] https://en.wikipedia.org/wiki/Newton%27s_method

[Wikipedia_7] https://en.wikipedia.org/wiki/Householder%27s_method

[Wikipedia_8] https://en.wikipedia.org/wiki/Rate_of_convergence

[Wikipedia_9] https://en.wikipedia.org/wiki/Fast_inverse_square_root

[Wikipedia_10] https://en.wikipedia.org/wiki/Bogosort

[Wikipedia_11] https://en.wikipedia.org/wiki/Square_root

[xkcd_1] http://www.xkcd.com/664/

[xkcd_2] https://xkcd.com/356/

[YBC7289] https://www.math.ubc.ca/~cass/Euclid/ybc/analysis.html

Polymorphic Comparisons

Polymorphic comparisons require much boilerplate. Robert Mill and Jonathan Coe introduce a template utility for such comparisons.

In this article, we discuss a class template utility called **PolyLessThan** that enables C++ programmers to rapidly develop and easily maintain a polymorphic comparator. **PolyLessThan** relies on the VISITOR pattern.

Ordering polymorphic objects

Suppose that we wish to maintain a collection of teachers and students resident in a school. Teachers are ordered by their employee number, whereas students are ordered sorted by their name. The ordering *within* a type is defined trivially by overloading the **<** operator, but comparisons *across* types (i.e., between **Residents**) are not catered for. The classes that define these entities are outlined in Listing 1.

Suppose next that we wish to maintain (i) a set of pointers to residents and (ii) a map of pointers to residents to their age in years. A standard solution that makes use of the **Containers** library is shown below:

```
set<const Resident*> set_residents;
map<const Resident*, int> map_resident_age;
```

Unless otherwise specified, a set or map will order these pointers according to their memory address, which may be unstable from one program execution to another and are obscure in relation to the object content, meaning that an iterator will traverse the objects in an unnatural and possibly unpredictable order. Consequently, one typically supplies a functor that provides a 'less-than' comparison operation via an additional

```
struct Resident
{
  ...
};

struct Teacher : Resident
{
  ...
  bool operator< (const Teacher& that) const
  {
    return that.ref < ref;
  }
  int ref;
};

struct Student : Resident
{
  ...
  bool operator< (const Student& that) const
  {
    return that.name < name;
  }
  string name;
};
```

Listing 1

```
struct TeacherLessThan
{
  bool operator() (
  const Teacher* pTeacher1,
  const Teacher* pTeacher2) const
  {
    return *pTeacher1 < *pTeacher2;
  }
}

set<const Teacher*,
  TeacherLessThan> set_teachers;
```

Listing 2

template argument. This is straightforward in the case of a derived type. Listing 2 shows an ordered set of **Teacher**s.

We now face the issue of how to compare **Resident**s – or pointers to them – in a natural, robust and extensible fashion.

By *natural*, we mean that the order should be defined in a content-wise fashion, based on datatypes and values, rather than in relation to a memory address or a hashcode. For instance, we could insist that $x < y$ for a teacher x and a student y.

By *robust*, we mean that reasoning about the types involved in the comparisons should work 'with the grain' of the C++ type system and not rely on support from type **enum**s, type casts or similar indicators. This we accomplish via use of the well-known VISITOR pattern, discussed below.

Finally, by *extensible*, we mean that it should be possible to derive new types from the base class and have them participate in comparisons (e.g., as set members or map keys) with minimal effort. For instance, we may wish to add an **AdminStaff** class, whose objects are sorted by start date.

Visitor pattern

The VISITOR pattern is a form of *dependency inversion*, which permits the definition of an operation outside of the class definitions, whilst retaining polymorphism via virtual dispatch [Gamma95]. Listing 3 shows how the code in Listing 1 can be fleshed out such that the **Resident** inheritance structure supports visiting.

To maintain a set of pointers to **Resident** ordered by content (as opposed to address or insertion order), we require a binary comparator

Robert Mill received his bachelor and Ph.D. degrees in Computer Science from the University of Sheffield. He now works in industrial process engineering as a mathematical developer, and retains an interest in machine learning and signal processing.

Jonathan Coe has been programming commercially for about 10 years. He has worked in the energy industry on process simulation and optimisation and is currently employed in the financial sector. You can contact Jonathan at jbcoe@me.com

Writing this code every time a new visitable inheritance hierarchy is defined is laborious

```
struct ResidentVisitor
{
  virtual ~ResidentVisitor() = default;
  virtual void Visit(const Teacher&) = 0;
  virtual void Visit(const Student&) = 0;
};

struct Resident
{
  virtual ~Resident() = default;
  virtual void Accept(ResidentVisitor& visitor)
    const = 0;
};

struct Teacher : Resident
{
  Teacher(int ref_) : ref(ref_) { }
  void Accept(ResidentVisitor& visitor)
    const override final
  {
    visitor.Visit(*this);
  }

  bool operator< (const Teacher& that) const
  {
    return ref < that.ref;
  }
  int ref;
};

struct Student : Resident
{
  Student(string name_) : name(name_) { }
  void Accept(ResidentVisitor& visitor) const
    override final
  {
    visitor.Visit(*this);
  }

  bool operator< (const Student& that) const
  {
    return name < that.name;
  }
  string name;
};
```

Listing 3

functor, such as that shown in Listing 4. How such a comparator should be defined is not immediately obvious, owing to the polymorphism of `Resident`.

Any visitor-based comparator must visit both `*pr1` and `*pr2` in order to establish their type. Within- or across-type comparisons can proceed once this information is available. However, writing this code every time a new visitable inheritance hierarchy is defined is laborious.

```
struct ResidentLessThan
{
  bool operator() (const Resident* pr1,
    const Resident* pr2) const
  {
    // Implementation...
  }
}

set<Resident*, ResidentLessThan> set_residents;
map<Resident*, Contact,
  ResidentLessThan> map_resident_contact;
```

Listing 4

Comparator Visitor

We propose the labour-saving class template `PolyLessThan` to facilitate sorting of visitable objects, defined in Listing 5.

The class template takes a pure virtual visitor base class as its first argument, followed by a complete variadic list of visitable types for the remainder of its arguments, such that types specified earlier in the list are less than those that come later. Listing 6 shows a `Resident` comparator that sorts `Teacher`s before `Student`s, along with an example of its deployment.

From the programmer's perspective, the task of defining a polymorphic comparator is accomplished entirely by this alias. If a new `Visit` clause is added to `ResidentVisitor`, then the `using` statement will not compile until the ordering over types is updated.

The implementation of the class template itself proceeds along similar lines to the inline visitor [Mill14, Coe15]. The private class `Impl` is templated on a particular item type and an ordering integer `N`. As each variadic argument is stripped off the list `TArgs`, `N` is incremented, and a new base class is defined; and this pattern recurses until all the arguments are consumed. The `Visit` functions are designed to be called up to twice.

- First, `*pt1` accepts `Impl` as a visitor. The invoked `Visit` member retains the pointer `pt1`, along with the template argument `N`, established at compile-time, which serves to enumerate the type. These are stored in protected members of the innermost `Impl` base class, `pt` and `n`, respectively. The `Impl` class is aware of the first invocation because a value of `0` for `n` serves as a sentinel.
- Second, `*pt2` accepts `Impl` as a visitor. When the control path enters the base class containing the `Visit` member, if the value for `N` matches that stored from the previous iteration, the types match, and the values are compared using the `<`operator particular to that sub-type. Otherwise, the values of `N` are themselves compared, which effects an ordering over types.

Although the logic underlying the template is recursive, this does not translate into recursive logic at runtime

```
template <class TVisitorBase, class ...TArgs>
class PolyLessThan
{

public:
  template <class T1, class T2>
  bool operator()(const T1* pt1,
    const T2* pt2) const
  {
    auto polyCompare = Impl<1, TArgs...>();
    pt1->Accept(polyCompare);
    pt2->Accept(polyCompare);
    return polyCompare.result;
  }

private:
  template <int N, class ...TInnerArgs>
  struct Impl : TVisitorBase
  {
    bool result = false;
  protected:
    int n = 0;
    const void* pt = nullptr;
  };
  template <int N, class TItem,
    class ...TInnerArgs>
  struct Impl<N, TItem, TInnerArgs...>
    : Impl<N+1, TInnerArgs...>
  {
    void Visit(const TItem &t) override final
    {
      if (this->n == 0)
      {
        this->n = N;
        this->pt = static_cast<const void *>(&t);
      }
      else if (this->n < N)
      {
        this->result = true;
      }
      else if (N < this->n)
      {
        this->result = false;
      }
      else
      {
        this->result = *static_cast<const TItem
          *>(this->pt) < t;
      }
    }
  };
```

Listing 5

```
  static_assert(
    !std::is_abstract<Impl<1, TArgs...>>::value,
    "Cannot compile polymorphic comparator: "
    "no concrete implementation for one or more "
    "Visit functions");
};
```

Listing 5 (cont'd)

```
using ResidentLessThan =
  PolyLessThan<ResidentVisitor,
  Teacher,
  Student>;

  auto student1 = Student("Jarvis");
  auto student2 = Student("Deborah");
  auto teacher1 = Teacher(1701);
  auto teacher2 = Teacher(24601);
  auto residents =
    set<const Resident*, ResidentLessThan>({
      &student1,
      &student2,
      &teacher1,
      &teacher2 });
```

Listing 6

Although the logic underlying the template is recursive, this does not translate into recursive logic at *runtime*; the outermost (i.e. the most derived) `Impl` class is simply an automated implementation of the visitor class that the consumer would need to write themselves without `PolyLessThan`. ■

References

[Coe15] Jonathan Coe, 'An Inline-variant-visitor with C++ Concepts', *Overload* 129, October 2015.

[Gamma95] E. Gamma et al., *Design Patterns*, Addison-Wesley, Longman, 1995.

[Mill14] Robert Mill and Jonathan Coe, 'Defining Visitors Inline in Modern C++' *Overload* 123, October 2014.

C++ Synchronous Continuation Passing Style

Direct and continuation passing styles differ. Nick Weatherhead explains a continuation passing style for synchronous data flow.

Imperative code can be viewed in terms of routines that in turn call sub-routines before passing control back to the point at which they were initiated and proceeding from there; this is known as Direct Style programming. Command shells often have the facility to pipe the output from one utility into the input of another. Adjoining self-contained modules in this way promotes loosely coupled functionality with a single purpose and well insulated state. For example, instrumentation can be conveniently implemented by intercepting a call, inspecting it and passing it on unaltered. It also enables content to be recorded and played to create or restore the state of a program.

Procedures can also transfer control forward if their product is a further procedure to call, hence the moniker Continuation Passing Style (CPS). Instead of a function having no visibility of where it returns and what is done with the result, it knows of the continuation called and the parameters passed to it. Different continuations can be chosen for different conditions including exceptional ones. They represent a program from a point forth. In doing so the call-stack is reified enabling computation to be captured and resumed. This article is an introductory exploration of their application in synchronous data flows, although they are equally adept as asynchronous callbacks.

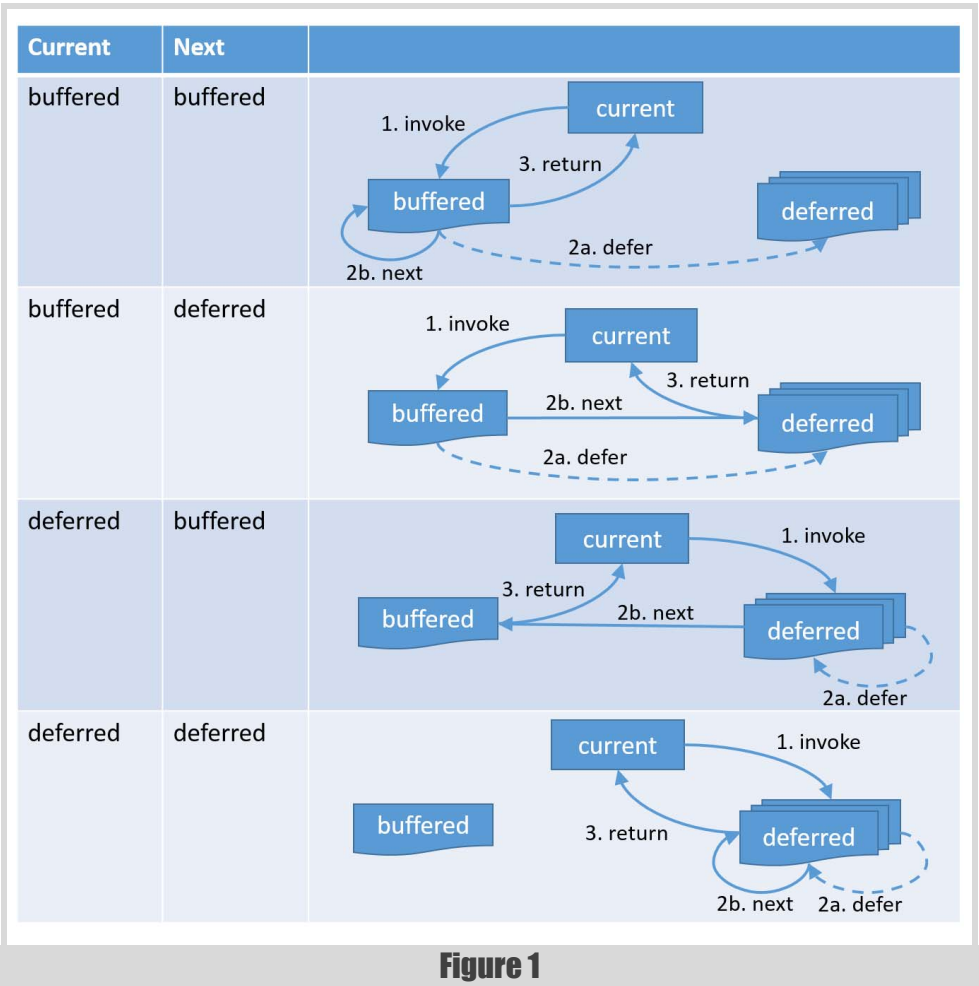


Figure 1

Trampoline style execution

Invoking a function places a frame containing variables local to it onto the runtime stack. Under normal circumstances this is removed once it returns. However, CPS logically flows forward so there are no returns in the traditional sense; instead a `return` is substituted by a function to `goto` next. In doing so, tail calls will accumulate until the stack overflows. Drawing an analogy to a trampoline this can be circumvented if, with each call, the stack cyclically goes up and comes back down again.

Nick Weatherhead Nick's first encounter with programming was copying lines of code from magazines into the now venerable family BBC B. His teacher persuaded him to take computer science during his first term of A-Levels. This led to many hours of puzzle solving and programming, a relevant degree and finally gainful employment within London's financial sector. You can contact Nick at weatherhead.nick@gmail.com

When parameters in the call before are not used again they can be replaced and the program counter sent back from whence it came. On other occasions the variables retained in outer frames are used once control returns. For example, the Quicksort is doubly recursive; repeatedly dividing partitions in two around a pivot point. Whilst the directives to partition one way, say left, need not be retained, those to the right need to be held until all the operations left of them have been completed. To accomplish this without use of the runtime stack they must be kept in auxiliary storage, nominally the heap, until required.

Figure 1 illustrates how a trampoline incorporating deferred computations can operate. *Current* points to a continuation to invoke and is repeatedly set as the result of its last operation and then called until the program aborts. *Buffered* continuations are written to a space set aside for their immediate use whilst *deferred* continuations are held in the heap for later. A continuation returns an opaque reference to one or other of these. So executing a *buffered* continuation results in it replacing itself or returning

Drawing an analogy to a trampoline this can be circumvented if, with each call, the stack cyclically goes up and comes back down again

```
#ifndef CHAIN_H
#define CHAIN_H
#include <iostream>

class chain {
public:
  constexpr const chain* operator( )( ) const {
    return onto_( *this ); }

protected:
  static void* const buffer_;
  using fn = const chain* ( & )( const chain& );
  explicit constexpr chain( fn onto )
  : onto_( onto ) { }
  constexpr chain( const chain& that )
  : onto_( that.onto_ ) { }

private:
  fn const onto_;
  const chain& operator=( const chain& );
};
…
```

Listing 1

one that had been deferred. Similarly a *deferred* continuation may return. or create one that is buffered. Executing either may result in the creation of one or more deferred operations. With each iteration the call stack unwinds and a loop returns the program counter back to where the aforementioned continuation is now ready to perform the next operation.

Quicksort example

Utilising the runtime stack is an elegant way to implement the Quicksort; however, its recursive nature means that this will grow. Adapting it to use continuations demonstrates the elimination of tail recursive calls, known as Tail Call Optimisation (TCO), and the utilisation of deferred computation. An implementation is shown below.

`Chain` (Listing 1) is the abstract base class for a continuation. It is composed of a single member, the function reference **`onto_`**, thereby avoiding the need for a virtual function table. This is initialised on construction and invoked via the function operator, which once called executes the current continuation and returns the subsequent one. The global pointer, **`buffer_`**, references space set aside for buffered continuations. This will later be sized to accommodate the largest one possible. Other strategies might arrange for the continuation object to be returned at the bottom of the call stack and proceed by advancing over it and on. While this may save space, manipulating the call stack adds complexity and must be done in a way that prevents corruption.

`Buffered` (Listing 2) glues the definition of an abstract continuation to a derived class's implementation. Static polymorphism is achieved by utilising the CURIOUSLY RECURRING TEMPLATE PATTERN [CRTP16]. Here the principle of inheriting derived behaviour is similar, but instead

```
…
template< class Chain, typename... Args >
class buffered : public chain {
public:
  static constexpr const Chain* create(
    Args... args ) {
    return new( chain::buffer_ ) Chain( args... );
  }

protected:
  constexpr buffered( ) : chain(
    static_cast< fn >( buffered::onto ) ) { }

private:
  static const chain* onto( const chain& that ) {
    const Chain& next =
      static_cast< const Chain& >( that );
    std::cerr << "buffered(" << next << ")\n";
    const chain* onto = next( );
    next.~Chain( ); return onto;
  }
};
…
```

Listing 2

of a class inheriting from a class template instantiation using itself, which in this case would be of the form **`chain< buffered >`**, it inherits from a regular class i.e. just **`chain`**. Thus **`chain`** is the base class from which both **`buffered`** and **`deferred`** objects derive and in turn means a **`chain`** pointer can be downcast to determine to which of these it refers. Variadic template arguments enable the creation of objects implementing a **`chain`** but which have different constructor signatures. Here a factory method, **`create`**, takes **`args`** to construct a derived continuation. This calls the derived class's constructor and **`placement new`** writes the object directly into the continuation buffer.

The **`onto`** function downcasts **`chain`** to the derived **`Chain`**; its function operator is then called. Before returning its destructor is explicitly called because of being placed in a buffer rather than on the call stack. It is these callbacks that are said to imitate '**`goto`** statements with arguments'. Whilst these jumps can make tracing code by hand more challenging, it need not make determining the execution path onerous. A continuation concerns itself with the content of the input rather than where it came from. Therefore, those that inspect input and output it unaltered can be injected between those that perform transformation without altering intent. Here, rather than injecting continuations, a **`stderr`** statement suffices for outputting trace. In production-like code, this could be replaced by categorised trace with each continuation having a bitmap of those categories to associate it with. This demonstrates that, unlike the traditional approach of peppering trace throughout a program, instrumentation can be achieved by observing what is passed between continuations.

As evidenced by eliminating tail recursion in Quicksort, inductive calls and non-local control flows are good candidates for continuations

```
…
template< class Chain, typename... Args >
class deferred : public chain {
public:
  static constexpr const chain* create(
    Args... args ) {
    return new deferred( args... );
  }

private:
  Chain const chain_;
  constexpr deferred( Args... args )
  : chain( deferred::onto ), chain_( args... ) { }
  static const chain* onto( const chain& that ) {
    const deferred& next =
      static_cast< const deferred& >( that );
    std::cerr << "deferred(" << next.chain_
              << ")\n";
    const chain* onto = next.chain_( );
    delete &next; return onto;
  }
};
#endif
```

Listing 3

Deferred (Listing 3) is the heap allocated equivalent of **buffered**. Static polymorphism enables a continuation, **chain_**, to be embedded within a deferred object. This is as opposed to maintaining a reference to one passed in, thus keeping allocation contiguous. As a **deferred** object is itself a continuation it can use its own function, **onto**, as its chained functor. When this is called it invokes **chain_** from the heap and the memory is freed by the encompassing object deleting itself. In this way it is a one-time computation responsible for its own allocation and deallocation.

Bound (Listing 4) uses a pair of pointers, **begin** and **end**, to demark an extent within an array. **Begin** points to the first element, and **end** just past the last element. From this its length can be calculated and there is an output operator that iterates over, and prints out, each element.

Terminate (Listing 5) prints the elements of an array and aborts a program. When instantiating a Quicksort it is passed in as a deferred operation, hence the **friend class** declaration so that a cached instance can access the **private** constructor. It is the first continuation on the stack of these deferred operations and thus the last in the chain of execution.

Quick (Listing 6) implements a rudimentary Quicksort taking the middle element of an array, placing elements lower than it to its left and higher than it to its right. The left and right partitions are then taken and repeatedly divided until they can't be partitioned any more, leaving the array in sorted order. Partitioning results in the left hand portion being written directly into the continuation buffer which is returned as the

```
#ifndef QUICK_H
#define QUICK_H
#include <cstdlib>
#include "chain.h"
template< typename T > struct bound {
  T* const begin_; T* const end_;
  constexpr bound( T* begin, T* end )
  : begin_( begin ), end_( end ) { }
  constexpr size_t length( ) const {
    return end_ - begin_; }
  friend std::ostream& operator<<(
    std::ostream& os, const bound& that ) {
    const T* itr = that.begin_; os << *itr;
    while( ++itr < that.end_ ) os << ' ' << *itr;
    return os;
  }
};
…
```

Listing 4

current continuation. The right hand portion references those already deferred, and adds itself to them, forming a stack of cached computation. If there are insufficient elements to partition then that most recently deferred is returned as the current continuation; and so it proceeds until the final deferred operation is reached and terminates the program. When pivoting left **quick** is created, by default, as a **buffered** object and when pivoting right as a **deferred** object. The **buffered** and **deferred friend** class declarations are requires so that **quick**'s **private** constructor can be accessed via each one's respective **create** factory method.

```
…
template< typename T > class terminator {
  friend class deferred< terminator, T*, T* >;
public:
  friend std::ostream& operator<<(
    std::ostream& os, const terminator& that ) {
    return os << "terminator(" <<
      that.bound_ << ")";
  }
  const chain* operator( )( ) const {
    std::cout << bound_ << "\n"; exit( 1 ); }

private:
  const bound< T > bound_;
  constexpr terminator( T* begin, T* end )
  : bound_( begin, end ) { }
};
…
```

Listing 5

```
...
template< class T > class quick
: public buffered< quick< T >, T*, T*,
    const chain* > {
  friend class buffered< quick, T*, T*,
     const chain* >;
  friend class deferred< quick, T*, T*,
     const chain* >;

public:
  friend std::ostream& operator<<(
    std::ostream& os, const quick& that ) {
    return os << "quick(" << that.bound_ << ")";
  }
  const chain* operator( )( ) const {
    size_t length = bound_.length( );
    if ( length < 2 ) return onto_;
    T  mid   = bound_.begin_[ length / 2 ];
    T* begin = bound_.begin_ - 1;
    T* end   = bound_.end_;
    for (;;) {
      while( *( ++begin ) < mid );
      while( *( --end   ) > mid );
      if ( begin >= end ) break;
      T temp = *begin; *begin = *end; *end = temp;
    }
    return quick::create( bound_.begin_, begin,
      deferred< quick, T*, T*, const chain* >::
        create( begin, bound_.end_, onto_ ) );
  }
  static constexpr const quick*
    create_with_terminator( T* begin, T* end ) {
    return quick::create( begin, end,
      deferred< terminator<T>, T*, T* >::
        create( begin, end ) );
  }

private:
  const bound< T > bound_;
  const chain* const onto_;
  constexpr quick( T* begin, T* end,
    const chain* onto )
  : bound_( begin, end ), onto_( onto ) { }
};
#endif
```

Listing 6

`Quick`'s constructor takes the continuation to move onto next as its last parameter. If there is no subsequent action to perform the program can exit, hence an overloaded constructor might be purposed to take just `begin` and `end` whilst defaulting the initialisation of `onto` to terminate. Nevertheless, when the compiler analyses the `create` factory method it continues to deduce that the constructor with more arguments, rather than those matching its signature, should be used. So, instead the call is wrapped in the aptly named `create_with_terminator`.

Finally, before starting the program (Listing 7) the continuation buffer is allocated of a size sufficient to store the largest continuation; in this case a `quick` sort operating on an array of integers. The `main` routine takes a space separated list of integer arguments from the command line and creates an array. The current continuation is defined as a `quick` sort on the entire array which, once complete, will execute `terminate`. Alternatively a continuation could be specified to go and use the sorted array in some other way. An infinite loop executes the program in trampoline style; the current continuation performing an operation and returning the next continuation in the chain.

Conclusion

As evidenced, by eliminating tail recursion in Quicksort, inductive calls and non-local control flows are good candidates for continuations. When flow is linear the active context is not revisited so can be overwritten with the next. This in combination with trampoline style execution ensures a compact stack. For flows parallel in nature the division of work, whether run separately or interleaved with others, needs to be captured. In the direct style the runtime stack implicitly suspends and resumes calls in the required order, but when using CPS these complexities are exposed and must be managed explicitly.

A detailed comparison of performance between direct and continuation passing styles isn't examined here. There is some overhead in calling a continuation over a regular function call. Unlike regular functions they are polymorphic requiring an indirection to execute them. There is also the auxiliary storage required to hold those deferred. Despite this only a marginal increase in execution time was observed when comparing the Quicksort presented with a recursive implementation. This could well be accentuated if, by specifying smaller packets of work, a proliferation of continuations occurred.

Whilst it takes time to become accustomed to CPS, it affords a way to express tasks and handle events via callbacks. An application programmer is likely to encounter its use for this purpose. CPS is also relevant in the implementation of programming languages and their compilers. Constructs can be defined, and conversely programs can be described, in terms of it [CPS16]. ■

References

[CRTP16] Curiously recurring template pattern, Wikipedia 2016.

[CPS16] Continuation-passing style, Wikipedia 2016.

Further reading

Andy Balham, Tail Call Optimisation in C++, *Overload* 109, June 2012.

Cristina Videira Lopes. *Exercises in programming style* Chapter 8: Kick Forward, Chapman and Hall/CRC, November 2015.

Acknowledgments

Many thanks to the *Overload* review team for their tips and observations which have benefited this article and my own understanding.

```
#include <cstddef>
#include "quick.h"

alignas( max_align_t )
char buffer[ sizeof( quick<int> ) ];
void* const chain::buffer_ = buffer;

int main( int argc, char* argv[] ) {
  int *data = ( int* ) calloc(
    --argc, sizeof( int ) );
  for( int i = 0; i < argc; ++i )
    data[i] = atoi( argv[i + 1] );
  const chain* current = quick<int>::
    create_with_terminator( data, &data[argc] );
  for (;;) current = ( *current )( );
}
```

Listing 7

Attacking Licensing Problems with C++

Software licenses are often crackable. Deák Ferenc presents a technique for tackling this problem.

From the early days of the commercialization of computer software, malicious programmers, also known as crackers, have been continuously nettling the programmers of the aforementioned software by constantly bypassing the clever licensing mechanisms they have implemented in their software, thus causing financial damages to the companies providing the software.

This trend has not changed in recent years: the cleverer the routines the programmers write, the more time is spent by crackers in invalidating the newly created routines, and in the end the crackers always succeed. For companies to be able to keep up with the constant pressure provided by the cracking community, they would need to constantly change their licensing and identification algorithms, but in practice this is not a feasible way to deal with the problem.

An entire industry has evolved around software protection and licensing technologies, where renowned companies offer advanced (and expensive) solutions to tackle this problem. The protection schemes range from using various resources such as hardware dongles, to network activation, from unique license keys to using complex encryption of personalized data – the list is long.

This article provides a short introduction to illustrate a very simple and naive licensing algorithm's internal workings. We will show how to bypass it in an almost real life scenario, and finally present a software based approach to mitigate the real problem by hiding the license checking code in a layer of obfuscated operations generated by the C++ template metaprogramming framework, which will make the life of the person wanting to crack the application a little bit harder. Certainly, if they are well determined, the code will still be cracked at some point, but at least we'll make it harder for them.

A naive licensing algorithm

The naive licensing algorithm is a very simple implementation that checks the validity of a license associated with the name of the user who purchased the associated software. It is *not* an industrial strength algorithm: it only has demonstrative power, while trying to provide insight to the actual responsibilities of a real licensing algorithm.

Since the license checking code is usually shipped with the software product in compiled form, I'll put in here both the generated code (in Intel x86 assembly) since that is what the crackers will see after a successful disassembly of the executable and the C++ code for the licensing algorithm. In order not to pollute precious page space with unintelligible binary code, I will restrict myself to including only the relevant bits of the code that naively determines whether a supplied license is valid or not, together with the C++ code that was used to generate the binary code.

Deák Ferenc Ferenc has wanted to be a better programmer for the last 15 years. Right now he tries to accomplish this goal by working at FARA (Trondheim, Norway) as a system programmer, and in his free time, by exploring the hidden corners of the C++ language in search for new quests. fritzone@gmail.com

```
static const char letters[] =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
bool check_license(const char* user,
                   const char* users_license)
{
  std::string license;
  size_t ll = strlen(users_license);
  size_t l = strlen(user), lic_ctr = 0;
  int add = 0;
  for (size_t i = 0; i < ll; i++)
    if (users_license[i] != '-')
      license += users_license[i];
  while (lic_ctr < license.length() ) {
    size_t i = lic_ctr;
    i %= l;
    int current = 0;
    while (i < l) current += user[i ++];
    current += add;
    add++;
    if (license[lic_ctr]
        != letters[current % sizeof letters])
      return false;
    lic_ctr++;
  }
  return true;
}
```

Listing 1

Listing 1 is the source code of the licensing algorithm.

The license which this method validates comes is in the form ABCD-EFGH-IJKL-MNOP, and there is an associated `generate_license` method which is presented as an appendix to this article.

Also, the naivety of this method is easily exposed by using the very proper name of `check_license` which immediately reveals to the want-to-be attacker where to look for the code checking the ... license. If you want to make harder for the attacker to identify the license checking method, I'd recommend either using some irrelevant names or just stripping all symbols from the executable as part of the release process.

The interesting part is the binary code of the method obtained via compilation of the corresponding C++ code (see Listing 2), which we obtained by compiling it with Microsoft Visual C++ 2015. I have compiled it in Release mode (with Debug information included for educational purposes) but it is intentionally *not* the Debug version, since we would hardly ship the debug version of the code to our customers.

I have also used the built-in debugger of the VS IDE to visualize the generated code next to the source, which facilitates a better understanding of the relation between the two of them.

Let's analyze it for a few moments. The essence of the validity checking happens at address `00FC15F8` where the comparison `cmp al, byte`

reverse engineering the license checking algorithm presented in the previous section would prove to be a highly challenging task

```
if (license[lic_ctr]
    != letters[current % sizeof letters])
  00FC15E4  lea       ecx,[license]
  00FC15E7  cmovae    ecx,dword ptr [license]
  00FC15EB  xor       edx,edx
  00FC15ED  push      1Bh
  00FC15EF  pop       esi
  00FC15F0  div       eax,esi
  00FC15F2  mov       eax,dword ptr [lic_ctr]
  00FC15F5  mov       al,byte ptr [ecx+eax]
  00FC15F8  cmp       al,byte ptr [edx+0FC42A4h]
  00FC15FE  jne       check_license+0DEh (0FC1625h)
  return false;
lic_ctr++;
  00FC1600  mov       eax,dword ptr [lic_ctr]
  00FC1603  mov       ecx,dword ptr [add]
  00FC1606  inc       eax
  00FC1607  mov       dword ptr [lic_ctr],eax
  00FC160A  cmp       eax,dword ptr [ebp-18h]
  00FC160D  jb        check_license+7Fh (0FC15C6h)
}
return true;
  00FC160F  mov       bl,1
  00FC1611  push      0
  00FC1613  push      1
  00FC1615  lea       ecx,[license]
  00FC1618  call
    std::basic_string<char,std::char_traits<char>,
    std::allocator<char> >::_Tidy (0FC1944h)
  00FC161D  mov       al,bl
}
  00FC161F  call      _EH_epilog3_GS (0FC2F7Ch)
  00FC1624  ret
  00FC1625  xor       bl,bl
  00FC1627  jmp       check_license+0CAh (0FC1611h)
```

Listing 2

`ptr [edx+0FC42A4h]` takes place (for those wondering, `edx` gets its value as being the remainder of the division at `00FC15F0`).

At this stage, the value of the `al` register is already initialized with the value of `license[lic_ctr]` and that is what is compared to the expected character. If it does not match, the code jumps to `0FC1625h` where the `bl` register is zeroed out (`xor bl, bl`) and from there the jump goes backward to `0FC1611h` to leave the method with the `ret` instruction found at `00FC1624`. Otherwise the loop continues.

The most common way of returning a value from a method call is to place the value in the `eax` register and let the calling code handle it, so before returning from the method the value of `al` is populated with the value of the `bl` register (via `mov al, bl` found at `00FC161D`).

Please remember that if the check discussed before did not succeed the value of the `bl` register was 0, but this `bl` was initialized to `1` (via `mov bl,1` at `00FC160F`) if the entire loop was successfully completed.

From the perspective of an attacker, the only thing that needs to be done is to replace the binary sequence of `xor bl,bl` with the binary code of `mov bl,1` in the executable. Since luckily these two are the same length (2 bytes), the crack is ready to be published within a few seconds.

Moreover, due to the simplicity of the implementation of the algorithm, a highly skilled cracker could easily create a key-generator for the application, which would be an even worse scenario as the cracker doesn't have to modify the executable. This means that further safety steps, such as integrity checks of the application, would all be executed correctly, but there would be a publicly available key-generator which could be used by anyone to generate a license-key without ever paying for it, or malicious salesmen could generate counterfeit licenses which they could sell to unsuspecting customers.

Here let's look at our C++ obfuscating framework.

The C++ obfuscating framework

The C++ obfuscating framework provides a simple macro-based mechanism, combined with advanced C++ template meta-programming techniques for relevant methods and control structures, to replace the basic C++ control structures and statements with highly obfuscated code which makes the reverse engineering of the product a complex and complicated procedure.

By using the framework, reverse engineering the license checking algorithm presented in the previous section would prove to be a highly challenging task due to the sheer amount of extra code generated by the framework engine.

The framework has adopted a familiar, BASIC-like, syntax to make the switch from real C++ source code to the macro language of the framework as easy and painless as possible.

Functionality of the framework

The role of the obfuscating framework is to generate extra code, while providing functionality which is expected by the user, with as few syntax changes to the language as possible.

The following functionalities are provided by the framework:

- wrap all values into a `valueholder` class thus hiding them from immediate access
- provide a BASIC-like syntax for the basic C++ control structures (`if`, `for`, `while` ...)
- generate extra code to achieve complex code, making it harder to understand
- randomize constant values in order to hide the information.

The value wrappers implement a limited set of operations which you can use to change the value of the wrapped variable

Debugging with the framework

Like every developer who has been there, we know that debugging complex and highly templated C++ code can sometimes be a nightmare. In order to avoid this nightmare while using the framework, we decided to implement a debugging mode.

To activate the debugging mode of the framework, define the `OBF_DEBUG` identifier before including the obfuscation header file. Please see the specific control structures for how the debugging mode alters the behaviour of the macro.

Using the framework

The basic usage of the framework boils down to including the header file providing the obfuscating functionality

```
#include "instr.h"
```

then using the macro pair `OBF_BEGIN` and `OBF_END` as delimiters of the code sequences that will be using obfuscated expressions.

For a more under-the-hood view of the framework, the `OBF_BEGIN` and `OBF_END` macros declare a `try-catch` block, which has support for returning values from the obfuscated current code sequence, and also provides support for basic control flow modifications such as the usage of `continue` and `break` emulator macros `CONTINUE` and `BREAK`.

Behind the scenes: OBF_BEGIN and OBF_END

`OBF_BEGIN` expands to:

```
#define OBF_BEGIN \
 try {obf::next_step __crv = \
    obf::next_step::ns_done; \
    std::shared_ptr<obf::base_rvholder> \
    __rvlocal;
```

and `OBF_END` becomes:

```
#define OBF_END } \
catch(std::shared_ptr<obf::base_rvholder>& r) { \
return *r; } catch (...) {throw;}
```

In order to support for 'returning' a value from the current obfuscated block we need a special variable `__rvlocal`. At later stages, this value will be populated with meaningful values as a result of executing the code of the `RETURN` macro (which will 'throw' a value with a type of `std::shared_ptr<obf::base_rvholder>`). The `OBF_END` will catch this specific value and handle it appropriately, while all other values thrown will be re-thrown in order to not to disturb the client code's exception handling.

Value and numerical wrappers

To achieve an extra layer of obfuscation, the integral numerical values can be wrapped in the macro `N()` and all integral numeric variables (`int`, `long`, ...) can be wrapped in the macro `V()` to provide an extra layer of obfuscation for doing the calculation operations. The `V()` value wrapper also can wrap individual array elements(`x[2]`), but not arrays (`x`) and also cannot wrap class instantiation values due to the fact that the macro expands to a reference holder object.

The implementation of the wrappers uses the link time random number generator provided by [Andrivet] and the values are obfuscated by performing various operations to hide the original value.

And here is an example for using the value and variable wrappers:

```
int a, b = N(6);
V(a) = N(1);
```

After executing the statement above, the value of `a` will be 1.

The value wrappers implement a limited set of operations which you can use to change the value of the wrapped variable. These are the compound assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=` and `^=`, and the post/pre-increment operations `--` and `++`. All of the binary operators (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `<<` and `>>`) are also implemented, so you can write `V(a) + N(1)` or `V(a) - V(b)`.

Also, the assignment operator to a specific type and from a different value wrapper is implemented, together with the comparison operators.

As the name implies, the value wrappers will wrap values by offering a behaviour similar to the usage of simple values, so be aware that variables which are `const` values can be wrapped into the `V()` wrapper but, as with real const variables, you cannot assign to them. So for example the following code will not compile:

```
const char* t = "ABC";
if( V(t[1]) == 'B')
{
  V( t[1] ) = 'D';
}
```

And the following

```
char* t = "ABC";
if( V(t[1]) == 'B')
{
  V( t[1] ) = 'D';
}
```

will be undefined behaviour because the compiler will highly probably allocate the string `"ABC"` in a constant memory area (although I would expect your compiler to choke heavily on this expression since it's not valid modern C++ anymore). To work with this kind of data, always use `char[]` instead of `char*`.

Behind the scenes of the implementation of numeric wrapping

The `N` macro is defined like the following:

```
#define N(a) (obf::Num<decltype(a), \
  obf::MetaRandom<__COUNTER__, 4096>:: value ^ \
  a>().get() ^ obf::MetaRandom<__COUNTER__ - 1, \
  4096>::value)
```

As a first step, let's consider that due to the implementation of [Andrivet] and the (more or less standard) `__COUNTER__` macro, the following will have the same value:

The value wrappers add an extra obfuscation layer to the values they wrap

```
template<typename T, T n> class Num final {
public:
  enum { value = ( (n & 0x01)
        | ( Num < T , (n >> 1)>::value << 1) )
  };
  Num() : v(0) {
    v = value ^  MetaRandom<32, 4096>::value;
  }
  T get() const {
    volatile T x = v ^ MetaRandom<32,
    4096>::value; return x;
  }
private:
  volatile T v;
};
```

Listing 3

```
struct ObfZero { enum {value = 0}; };
struct ObfOne { enum {value = 1}; };

#define OBF_ZERO(t) template <> struct Num<t,0>
final : public ObfZero { t v = value; };

#define OBF_ONE(t) template <> struct Num<t,1>
final : public ObfOne { t v = value; };

#define OBF_TYPE(t) OBF_ZERO(t) OBF_ONE(t)
OBF_TYPE(int) // And for all other integral types
```

Listing 4

```
  int n;
  OBF_BEGIN
   n = N(42);
002A5F74  mov         dword ptr [ebp-4],0
002A5F7B  mov         dword ptr [ebp-4],78Ch
002A5F82  mov         eax,dword ptr [ebp-4]
002A5F85  xor         eax,0E8Fh
002A5F8A  mov         dword ptr [ebp-4],eax
002A5F8D  mov         eax,dword ptr [ebp-4]
002A5F90  xor         eax,929h
    OBF_END
```

Listing 5

```
obf::MetaRandom<__COUNTER__, 4096>::value
obf::MetaRandom<__COUNTER__ - 1, 4096>::value)
```

Now, taking the `obf::Num` class into view, we have Listing 3, where the iteration of the templates is finalized by Listing 4.

The `Num` class tries to add some protection by adding some extra xor operations to the use of a simple number, thus turning a simple numeric assignment into several steps of assembly code (Visual Studio 2015 generated the code Listing 5 in Release With Debug Info mode).

However, please note the several `volatile` variables ... which are required to circumvent today's extremely clever optimizing compilers. If we remove the `volatile` from the variables, the compiler is clever enough to guess the value I wanted to obfuscate, so ... there goes the obfuscation.

Behind the scenes of the implementation of variable wrapping

When we are not building the code in debugging mode, the macro `V` expands to the following C++ nightmare:

```
#define MAX_BOGUS_IMPLEMENTATIONS 3

#define V(a) ([&]() \
  {obf::extra_chooser<std::remove_reference \
    <decltype(a)>::type, \
    obf::MetaRandom<__COUNTER__,\
    MAX_BOGUS_IMPLEMENTATIONS>::value > \
    ::type _JOIN(_ec_,__COUNTER__)(a);\
    return obf::stream_helper();}() << a)
```

So let's dissect it in order to understand the underlying operations.

The value wrappers add an extra obfuscation layer to the values they wrap, by performing an extra addition, an extra subtraction or an extra xor operation on the value itself. This is picked randomly when compilation happens by the `extra_chooser` class, which is like:

```
template <typename T, int N>
class extra_chooser
{
  using type = basic_extra;
};
```

and is helped by the following constructs:

```
#define DEFINE_EXTRA(N,implementer) template \
 <typename T> struct extra_chooser<T,N> { \
 using type = implementer<T>; }

 DEFINE_EXTRA(0, extra_xor);
 DEFINE_EXTRA(1, extra_substraction);
 DEFINE_EXTRA(2, extra_addition);
```

which are the actual definition of the classes for the extra operations, which in their turn look like Listing 6, where the extra addition and subtraction are also very similar.

The next thing we observe is that an object of this kind (extra bogus operation chooser) is defined in a lambda function for the variable we are wrapping. The variable name for this is determined by `_JOIN(_ec_,__COUNTER__)(a)`, where `_JOIN` is just a simple joiner macro:

```
#define _JOIN(a,b) a##b
```

Upon creation and destruction of this `extra_chooser` object, the value of the object will remain unchanged; however, extra code will be generated by the compiler (thanks to the numerous `volatile` modifiers

```
template <class T>
class extra_xor final : public basic_extra
{
public:
  extra_xor(T& a) : v(a)
  {
    volatile T lv
      = MetaRandom<__COUNTER__, 4096>::value;
    v ^= lv;
  }
  virtual ~extra_xor()
  {
    volatile T lv
      = MetaRandom<__COUNTER__ - 1, 4096>::value;
    v ^= lv;
  }
private:
  volatile T& v;
};
```

Listing 6

found in the extra operation classes, otherwise the compiler would 'cheat' again and just 'skip' our obfuscation). This is actually an extensible interface, so you can use it to define your own class for bogus operations using the `DEFINE_EXTRA` macro (and increase the `MAX_BOGUS_IMPLEMENTATIONS` as required).

Now, back to the lambda because it plays an important role. The lambda returns an object of type `obf::stream_helper()`, which is basically an empty class (`class stream_helper {};`), but the role of the lambda is still not done. As we can see in the macro, the lambda is executed and into its result (the `obf::stream_helper()` object) we stream the parameter of the macro (`<< a`). This gives control to the following operator:

```
template <typename T>
refholder<T> operator << (stream_helper, T& a)
{
  return refholder<T>(a);
}
```

providing us with a controversial class, `refholder` (Listing 7).

This class has all the support for the basic operations you can execute on a variable either via the member operators (defined explicitly or via the

```
template <typename T>
class refholder final
{
public:
  refholder() = delete;
  refholder(T& pv) : v(pv) {}
  refholder(T&&) = delete;
  ~refholder() = default;
  refholder<T>& operator = (const T& ov) {
    v = ov; return *this;
  }
  refholder<T>& operator
    = (const refholder<T>& ov) {
    v = ov.v; return *this;
  }
  bool operator == (const T& ov) {
    return !(v ^ ov);
  }
  bool operator != (const T& ov) {
    return !operator ==(ov);
  }
  COMPARISON_OPERATOR(>=)
  COMPARISON_OPERATOR(<=)
  COMPARISON_OPERATOR(>)
  COMPARISON_OPERATOR(<)
  operator T() {return v;}
  refholder<T>& operator++() {
    ++ v; return *this;
  }
  refholder<T>& operator--() {
    -- v; return *this;
  }
  refholder<T> operator++(int) {
    refholder<T> rv(*this); operator ++();
    return rv;
  }
  refholder<T> operator--(int) {
    refholder<T> rv(*this); operator --();
    return rv;
  }
  COMP_ASSIGNMENT_OPERATOR(+)
  COMP_ASSIGNMENT_OPERATOR(-)
  COMP_ASSIGNMENT_OPERATOR(*)
  COMP_ASSIGNMENT_OPERATOR(/)
  COMP_ASSIGNMENT_OPERATOR(%)
  COMP_ASSIGNMENT_OPERATOR(<<)
  COMP_ASSIGNMENT_OPERATOR(>>)
  COMP_ASSIGNMENT_OPERATOR(&)
  COMP_ASSIGNMENT_OPERATOR(|)
  COMP_ASSIGNMENT_OPERATOR(^)
private:
  volatile T& v;
};
```

Listing 7 (cont'd)

macro `COMP_ASSIGNMENT_OPERATOR`) either defined via the `DEFINE_BINARY_OPERATOR` macro which defines binary operators for `refholder` classes. In cases when the variable wrapping is done on constant variables, there are specializations of this template class for constant `T`s. There are various arguments against the construct of storing

The sheer amount of extra code generated for a simple assignment is overwhelming

references as class members [Stackoverflow]; however, I consider this situation to be a reasonably safe one which can be exploited for this specific reason. So, here (Listing 8) comes a piece of generated assembly code for a very simple expression.

The sheer amount of extra code generated for a simple assignment is overwhelming.

Control structures of the framework

The basic control structures which are familiar from C++ are made available for immediate use by the developers by means of macros, which expand into complex templated code.

They are meant to provide the same functionality as the standard C++ keyword they are emulating, and if the framework is compiled in DEBUG mode, most of them actually expand to the C++ control structure itself.

Decision making

When there is a need in the application to take a decision based on the value of a specific expression, the obfuscated framework offers the familiar **`if-then-else`** statement for the developers in the form of the **`IF-ELSE-ENDIF`** construct.

```
    int n;
    OBF_BEGIN
        V(n) = N(42);
00048466  mov         dword ptr [ebp-8],0
0004846D  mov         dword ptr [ebp-8],97Ch
00048474  push        esi
00048475  mov         esi,dword ptr [ebp-8]
00048478  mov         dword ptr [ebp-8],48Bh
0004847F  xor         esi,0DC4h
00048485  mov         eax,dword ptr [ebp-8]
00048488  add         eax,dword ptr [n]
0004848B  mov         dword ptr [n],eax
0004848E  mov         dword ptr [ebp-8],48Bh
00048495  mov         eax,dword ptr [ebp-8]
00048498  sub         dword ptr [n],eax
0004849B  lea         eax,[n]
0004849E  push        eax
0004849F  push        dword ptr [ebp-8]
000484A2  lea         eax,[ebp-0Ch]
000484A5  push        eax
000484A6  call        obf::operator<<<int>
(0414C9h)
000484AB  add         esp,0Ch
000484AE  xor         esi,492h
000484B4  mov         eax,dword ptr [eax]
000484B6  mov         dword ptr [eax],esi
    OBF_END
```

Listing 8

The IF statement

For checking the true-ness of an expression the framework offers the **`IF`** macro which has the following form:

```
IF (expression)
....statements
ELSE
....other statements
ENDIF
```

where the **`ELSE`** is not mandatory, but the **`ENDIF`** is, since it indicates the end of the **`IF`** block's statements.

And here is an example for the usage of the **`IF`** macro.

```
IF( V(a) == N(9) )
  V(b) = a + N(5);
ELSE
  V(a) = N(9);
  V(b) = a + b;
ENDIF
```

Due to the way the **`IF`** macro is defined, it is not necessary to create a new scope between the **`IF`** and **`ENDIF`**; it is automatically defined and all variables declared in the statements between **`IF`** and **`ENDIF`** are destroyed.

Since the evaluation of the **`expression`** is bound to the execution of a hidden (well, at least from the outer world) lambda, unfortunately it is not possible to declare variables in the **`expression`** so the following:

```
IF( int x = some_function() )
```

is not valid, and will yield a compiler error. This is partially intentional, since it gives that extra layer of obfuscation required to hide the operations done on a variable in a nameless lambda somewhere deep in the code.

In cases when debugging mode is active, the **`IF-ELSE-ENDIF`** macros are defined to expand to the following statements:

```
#define IF(x)  if(x) {
#define ELSE   } else {
#define ENDIF  }
```

Implementation of the IF construct

The **`IF`** macro expands to the following:

```
#define IF(x) { \
  std::shared_ptr<obf::base_rvholder> __rvlocal;\
  obf::if_wrapper(( [&]()->bool{ return (x); \
  })).set_then( [&]() {
```

the **`ELSE`** macro expands to:

```
#define ELSE return __crv;}).set_else( [&]() {
```

and the **`ENDIF`** will give:

```
#define ENDIF return __crv;}).run(); }
```

```
{
  std::shared_ptr<obf::base_rvholder> __rvlocal;
  obf::if_wrapper( ([&]()->bool
  {
    return (n == 42);
  }) )
  .set_then( [&]()
  {
    n = 43;
    return __crv;
  })
  .set_else( [&]()
  {
    n = 44;
    return __crv;
  })
  .run();
}
```

Listing 9

So to wrap it all up, the following code:

```
IF( n == 42)
  n = 43;
ELSE
  n = 44;
ENDIF
```

will expand to Listing 9.

Now let's examine the `if_wrapper` class (Listing 10).

It is very clear why we needed the lambda created by the `IF` macro `(([&]()->bool { return (n == 42); }))`: we needed to create an object of type `class bool_functor` from it, which will give us the true-ness of the if condition. The bool functor class looks like Listing 11, where the important part is the `bool run()` – which in fact runs the condition and returns its true-ness.

The two branches of the `if` are represented by the member variables `std::unique_ptr<next_step_functor_base> thens;` `std::unique_ptr<next_step_functor_base> elses;` and they behave very similarly to the conditional.

The `run()` method of the `if_wrapper` class firstly checks the condition and then, depending on the presence of the `then` and `else` branches, executes the required operations.

Support for looping

There are times when every application needs to iterate over a set of values, so I tried to re-implement the basic loop structures used in C++: the `for` loop, the `while` and the `do-while` have been reincarnated in the framework.

```
class if_wrapper final
{
public:
  template<class T>
  if_wrapper(T lambda) {
    condition.reset(new bool_functor<T>(lambda));}
  void run()
  {
    if(condition->run()) { if(thens) {
      thens->run();
    }}
    else { if(elses) {
      elses->run();
    }}
  }
  ~if_wrapper() noexcept = default;
  template<class T>
  if_wrapper& set_then(T lambda)
  {
    thens.reset(new next_step_functor<T>(lambda));
    return *this;
  }
  template<class T>
  if_wrapper& set_else(T lambda)
  {
    elses.reset(new next_step_functor<T>(lambda));
    return *this;
  }
private:
  std::unique_ptr<bool_functor_base> condition;
  std::unique_ptr<next_step_functor_base> thens;
  std::unique_ptr<next_step_functor_base> elses;
};
```

Listing 10

```
struct bool_functor_base
{
  virtual bool run() = 0;
};

template <class T>
struct bool_functor final : public
bool_functor_base
{
  bool_functor(T r) : runner(r) {}
  virtual bool run() {return runner();}

private:
  T runner;
};
```

Listing 11

The FOR statement

The macro provided to imitate the **for** statement is:

```
FOR(initializer, condition, incrementer)
.... statements
ENDFOR
```

Please note that, since **FOR** is a macro, it should use **,** (comma) not the traditional **;** which is used in the standard C++ **for** loops, and do not forget to include your **initializer**, **condition** and **incrementer** in parentheses if they are expressions which have **,** (comma) in them.

The **FOR** loops should be ended with and **ENDFOR** statement to signal the end of the structure. Here is a simple example for the **FOR** loop.

```
FOR(V(a) = N(0), V(a) < N(10), V(a) += 1)
  std::cout << V(a) << std::endl;
ENDFOR
```

The same restriction concerning the variable declaration in the **initializer** as in the case of the **IF** applies for the **FOR** macro too, so it is not valid to write:

```
FOR(int x=0, x<10, x++)
```

and the reasons are again the same as presented above.

In a debugging session, the **FOR-ENDFOR** macros expand to the following:

```
#define FOR(init,cond,inc) for(init;cond;inc) {
#define ENDFOR }
```

The WHILE loop

The macro provided as replacement for the **while** is:

```
WHILE(condition)
....statements
ENDWHILE
```

The **WHILE** loop has the same characteristics as the **IF** construct and behaves the same way as you would expect from a well-mannered **while** statement: it checks the condition at the top, and executes the statements repeatedly as long as the given condition is true. Here is an example for **WHILE**:

```
V(a) = 1;
WHILE( V(a)  < N(10) )
  std::cout << "IN:" << a<< std::endl;
  V(a) += N(1);
ENDWHILE
```

Unfortunately the **WHILE** loop also has the same restrictions as the **IF**: you cannot declare a variable in its condition.

If compiled in debugging mode, the **WHILE** evaluates to:

```
#define WHILE(x) while(x) {
#define ENDWHILE }
```

The REPEAT-AS_LONG_AS construct posing as do-while

Due to the complexity of the solution, the familiar **do-while** construct of the C++ language had to be renamed a bit, since the **WHILE** 'keyword' was already taken for the benefit of the **while** loop, so I created the **REPEAT-AS_LONG_AS** keywords to achieve this goal.

This is the syntax of the **REPEAT-AS_LONG_AS** construct:

```
REPEAT
....statements
AS_LONG_AS( expression )
```

This will execute the statements at least once and then, depending on the value of the **expression**, either will continue the execution, or will stop and exit the loop. If the expression is **true**, it will continue the execution from the beginning of the loop; if the expression is **false**, execution will stop and the loop will be exited.

And here is an example:

```
REPEAT
  std::cout << a << std::endl;
  ++ V(a);
AS_LONG_AS( V(a) != N(12) )
```

When debugging, the **REPEAT - AS_LONG_AS** construct expands to the following:

```
#define REPEAT   do {
#define AS_LONG_AS(x) } while (x);
```

Implementation of the looping constructs

The logic and design of the looping constructs are very similar to each other. They behave very similarly to **IF** and each of them uses the same building blocks. There are the wrapper classes (**for_wrapper**, **repeat_wrapper**, **while_wrapper**), each of them with their functors for verifying the condition, and the steps to be executed.

The implementation in each of the **run()** method of the wrapper class follows the logic of the keyword it tries to emulate, with the exception that the commands are wrapped into a **try - catch** to enable **BREAK** and **CONTINUE** to function properly. Let's see for example the **run()** of the **for** wrapper:

```
void run()
{
  for( initializer->run(); condition->run();
    increment->run())
  {
    try
    {
      next_step c = body->run();
    }
    catch(next_step& c)
    {
      if(c == next_step::ns_break) break;
      if(c == next_step::ns_continue) continue;
    }
  }
}
```

Altering the control flow of the application

Sometimes there is a need to alter the execution flow of a loop. C++ supports this operation by providing the **continue** and **break** statements. The framework offers the **CONTINUE** and **BREAK** macros to achieve this goal.

The CONTINUE statement

The **CONTINUE** statement will skip all statements that follow it in the body of the loop, thus altering the flow of the application.

Here is an example for the **CONTINUE** used in a **FOR** loop:

```
FOR(a = 0, a < 5, a++)
  std::cout << "counter before=" << a
  <<   std::endl;
  IF(a == 2)
    CONTINUE
  ENDIF
  std::cout << "counter after=" << a
  << std::endl;
ENDFOR
```

and the equivalent **WHILE** loop:

```
a = 0;
WHILE(a < 5)
  std::cout << "counter before=" << a
  << std::endl;
  IF(a == 2)
    a++;
    CONTINUE
  ENDIF
  std::cout << "counter after=" << a
  << std::endl;
  a++;
ENDWHILE
```

Neither of these should print out the **counter after=2** text.

The BREAK statement

The **BREAK** statement terminates the loop statement it resides in and transfers execution to the statement immediately following the loop.

Here is an example for the **BREAK** statement used in a **FOR** loop:

```
FOR(a = 0, a < 10, a++)
  std::cout << "counter=" << a << std::endl;
  IF(a == 1)
    BREAK
  ENDIF
ENDFOR
```

This loop will print **counter=0** and **counter=1** then it will leave the body of the loop, continuing the execution after the **ENDFOR**.

The RETURN statement

As expected, the **RETURN** statement returns the execution of the current function and will return the specified value to the caller function. Here is an example of returning 42 from a function:

```
int some_fun()
{
  OBF_BEGIN
    RETURN(42)
  OBF_END
}
```

With the introduction of **RETURN**, an important issue arose: the obfuscation framework does not support the use of **void** functions, so the following code will not compile:

```
void void_test(int& a)
{
  OBF_BEGIN
    IF(V(a) == 42)
      V(a) = 43;
    ENDIF
  OBF_END
}
```

This is a seemingly annoying feature, but it can easily be fixed by simply changing the return type of the function to any non-void type. The reason is that the **RETURN** macro and the underlying C++ constructs should handle a wide variety of returnable types in a manner which can be handled easily by the programmer without causing confusion.

Implementation of CONTINUE, BREAK and RETURN

These keywords give the following when not compiled in debug mode:

```
#define BREAK __crv = obf::next_step::ns_break; \
   throw __crv;

#define CONTINUE __crv = \
   obf::next_step::ns_continue; throw __crv;

#define RETURN(x) __rvlocal.reset\
  (new obf::rvholder<std::remove_reference\
  <decltype(x)> ::type>(x,x)); throw __rvlocal;
```

BREAK and **CONTINUE** offer no surprises in the implementation and they comply to the expectation that has been formulated in the looping constructs: they throw a specific value, which is then caught in the local loop of the implementation, which handles it accordingly.

However, **RETURN** is a different kind of beast.

It initializes the **__rvlocal** (the local return value) to the returned value and then throws it for the **catch** which is to be found in the **OBF_END** macro, which in its turn handles it correctly.

As you can see, there are three evaluations of the **x** macro parameter. To avoid unwanted behaviour from your application, do not use expressions which might turn out to be dangerous, such as **RETURN (x++);**, which will give a three-times increment to your variable and undefined behaviour.

The **rvholder** class has the body shown in Listing 12.

As you can see there is a redundant **equals** method in the base class, and this is due to the fact that during development of the framework, the Visual Studio compiler constantly crashed due to some internal error in the implementation of the **CASE** construct, and it always reported the error in the **operator ==** of the base class. In order to make it work, I have added the extra **equals** member.

The CASE statement

When programming in C++, the **switch-case** statement comes in handy when there is a need to avoid long chains of **if** statements. The obfuscation framework provides a similar construct, although not exactly a functional and syntactical copy of the original **switch-case** construct.

Here is the **CASE** statement:

```
CASE (<variable>)
  WHEN(<value>) [OR WHEN(<other_value>)] DO
  ....statements
  ....[BREAK]
  DONE
  [DEFAULT
  ....statements
  DONE]
ENDCASE
```

The functionality is very similar to the well-known **switch-case** construct, the main differences are:

1. It is possible to use non-numeric, non-constant values (variables and strings) for the **WHEN** due to the fact that all of the **CASE** statement is wrapped up in a templated, lambdaized, well-hidden from the

```
struct base_rvholder
{
  virtual ~base_rvholder() = default;

  template<class T>
  operator T () const
  {
    return *reinterpret_cast<const T*>(get());
  }
  template<class T>
  bool operator == (const T& o) const
  {
    return o == operator T ();
  }
  template<class T>
  bool equals(const T& o) const
  {
    return o ==
      *reinterpret_cast<const T*>(get());
  }
  virtual const void* get() const = 0;
};

template<class T>
class rvholder : public base_rvholder
{
public:
  rvholder(T t, T c) :
    base_rvholder(), v(t), check(c) {}
  ~rvholder() = default;
  virtual const void* get() const override
  {
    return reinterpret_cast<const void*>(&v);
  }
private:
  T v;
  T check;
};
```

Listing 12

```
std::string something = "D";
std::string something_else = "D";

CASE (something)
  WHEN("A") OR WHEN("B") DO
    std::cout <<"Hurra, something is "
    << something << std::endl;
    BREAK;
  DONE

  WHEN("C") DO
    std::cout <<"Too bad, something is "
    << something << std::endl;
    BREAK;
  DONE

  WHEN(something_else) DO
    std::cout <<"Interesting, something is "
    << something_else << std::endl;
    BREAK;
  DONE

  DEFAULT
    std::cout << "something is neither A, B or C,"
    " but:" << something <<std::endl;
  DONE
ENDCASE
```

Listing 13

outside world, construct. Be careful with this extra feature when using the debugging mode of the library because the **CASE** macro expands to the standard **case** keyword.

2. It is possible to have multiple conditions for a **WHEN** label joined together with **OR**.

The fall through behaviour of the **switch** construct which is familiar to C++ programmers was kept, so there is a need to put in a **BREAK** statement if you wish the operation to stop after entering a branch.

Listing 13 is an example for the **CASE** statement.

In cases when the framework is used in debugging mode, the macros expand to the following statements:

```
#define CASE(a) switch (a) {
#define ENDCASE }
#define WHEN(c) case c:
#define DO {
#define DONE }
#define OR
#define DEFAULT default:
```

Implementation of the CASE construct

Certainly, the most complex of all constructs is the **CASE** one. Just the number of macros supporting it is huge:

```
#define CASE(a) try { \
  std::shared_ptr<obf::base_rvholder> __rvlocal;\
  auto __avholder = a; \
  obf::case_wrapper<std::remove_reference \
  <decltype(a)>::type>(a).

#define ENDCASE run(); } \
  catch(obf::next_step& cv) {}

#define WHEN(c)\
  add_entry(obf::branch<std::remove_reference\
  <decltype(__avholder)>::type> \
  ( [&,__avholder]() -> \
  std::remove_reference<decltype(__avholder)>\
  ::type { \
  std::remove_reference<decltype(__avholder)>\
  ::type __c = (c); return __c;} )).

#define DO add_entry( obf::body([&](){

#define DONE return \
  obf::next_step::ns_continue;})).

#define OR join().

#define DEFAULT add_default(obf::body([&](){
```

Let's dive into it.

The **case_wrapper** name should be already familiar from the various wrappers, but for **CASE**, the real workhorse is the **case_wrapper_base** class. The **case_wrapper** class is necessary in order to make **CASE** selection on **const** or non **const** objects possible, so the **case_wrapper** classes just derive from **case_wrapper_base** and specialize on the **const**ness of the **CASE** expression. Please note that the **CASE** macro also evaluates more than one the **a** parameters, so writing **CASE(x++)** will lead to undefined behaviour.

The **case_wrapper_base** class looks like Listing 14.

The **const CT check;** is the expression that is being checked for the various case branches. Please note the **add_entry** and **add_default** methods, together with the **join()** method which allow chaining of expressions and method calls on the same object. The **std::vector<const case_instruction*> steps;** is a cumulative container for all the branch condition expressions and bodies (code which is executed in a branch). This will introduce more complex code at a later stage; however, it was necessary to have these two joined in the same container in order to allow behaviour as similar to the original way the C++ **case** works as possible.

The inner mechanism of the **CASE** depends on the following classes:

1. The **obf::case_instruction** class, which acts as a basic class for:
2. **obf::branch** and
3. **obf::body** classes.

The **obf::branch** class is the class which gets instantiated by the **WHEN** macro in a call to the **add_entry** method of the **case_wrapper** object created by **CASE**. Its role is to act as the condition chooser, and it looks like Listing 15.

```
template <class CT>
class case_wrapper_base
{
public:
  explicit case_wrapper_base(const CT& v) :
check(v), default_step(nullptr) {}
  case_wrapper_base& add_entry(const
case_instruction& lambda_holder) {
    steps.push_back(&lambda_holder);
    return *this;
  }
  case_wrapper_base& add_default(const
    case_instruction& lambda_holder) {
      default_step = &lambda_holder;
      return *this;
    }
  case_wrapper_base& join() {
    return *this;
  }
  void run() const ; // body extracted from here,
      // see later in the article for the
      // description of it
private:
  std::vector<const case_instruction*> steps;
  const CT check;
  const case_instruction* default_step;
};
```

Listing 14

```
template<class CT>
class branch final : public case_instruction
{
public:
  template<class T>
  branch(T lambda)
  {
    condition.reset(new any_functor<T>(lambda));
  }
  bool equals(const base_rvholder& rv, CT lv)
const
  {
    return rv.equals(lv);
  }
  virtual next_step execute(const base_rvholder&
against) const override
  {
    CT retv;
    condition->run(const_cast<void*>
      (reinterpret_cast<const void*>(&retv)));
    return equals(against,retv) ?
      next_step::ns_done : next_step::ns_continue;
  }
private:
  std::unique_ptr<any_functor_base> condition;
};
```

Listing 15

The **WHEN** macro has a more or less confusing lambda declaration which includes the local **__avholder** as being passed in by value. This is again due to the fact that various compilers decided to not to compile the same source code in the same way... well, some of them had a coup and bluntly declined to compile what the others already digested, that's why the ugly solution came into existence.

The code that is executed upon entering a branch (including the default branch) is created by the **DO** and the **DEFAULT** macros. They both create an instance of the **obf::body** class: **DO** adds it to the steps of the case wrapper class, and **DEFAULT** calls the **add_default** member in order to specify a default branch. The **oft::body** class is much simpler, just a few lines (see Listing 16).

The most interesting (and longest) part of the **case** implementation is the **run()** method, presented here (in a somewhat stripped manner – I have removed all the security checks in order to have presentable code considering its length) – see Listing 17.

As a first step the code looks for the first branch which satisfies the condition (if **(*it)->execute(rvholder<CT>(check,check));** returns **next_step::ns_done** it means it has found a branch satisfying the **check**). In this case it skips all the other conditions for this branch and starts executing the code for all the **ofb::body** classes that are in the object. In case a **BREAK** statement was issued while executing the bodies the code will throw and the **catch** in **ENDCASE** (**catch(obf::next_step& cv**) will swallow it, and will return the execution to the normal flow.

```
class body final : public case_instruction
{
public:
  template<class T>
  body(T lambda)
  {
    instructions.reset
      (new next_step_functor<T>(lambda));
  }
  virtual next_step execute
    (const base_rvholder&) const override
  {
    return instructions->run();
  }
private:
  std::unique_ptr<next_step_functor_base>
    instructions;
};
```

Listing 16

The last resort is that if we have a **default_step** and we are still in the body of the run (no-one issued a **BREAK** command) it also executes it.

And with this we have presented the entire framework, together with implementation details, and now we are ready to catch up with our initial goal.

The naive licensing algorithm revisited

Now that we are aware of a library that offers code obfuscation without too many headaches from our side (at least, this was the intention of the author) let's re-consider the implementation of the naive licensing algorithm using these new terms (see Listing 18).

Indeed, it looks a little bit more 'obfuscated' than the original source, but after compilation it adds a great layer of extra code around the standard logic, and the generated binary is much more cumbersome to understand than the one 'before' the obfuscation. And due to the sheer size of the generated assembly code, we simply omit publishing it here.

Disadvantages of the framework

Those who dislike the usage of CAPITAL letters in code may find the framework to be annoying. As presented in [Wakely14] this almost feels like the code is shouting at you. However, for this particular use case, I intentionally made it like this because of the need to have familiar words that a developer can instantly connect with (because the lower case words are already keywords), and also to subscribe to the C++ rule that macros should be upper case.

This brings us back to the swampy area of C++ and macros. There are several voices whispering loudly that macros have nothing to do in C++ code, and there are several voices echoing back that macros, if used

```
void run() const
{
  auto it = steps.begin();
  while(it != steps.end()) {
    next_step enter
      = (*it)->execute(rvholder<CT>(check,check));
    if(enter == next_step::ns_continue) {
      ++it;
    }
    else {
      while(! dynamic_cast<const body*>(*it)
        && it != steps.end() )
      {
        ++it;
      }
      // found the first body.
      while(it != steps.end()) {
        if(dynamic_cast<const body*>(*it))
        {
          (*it)->execute(rvholder<CT>
            (check,check));
        }
        ++it;
      }
    }
  }
  if(default_step) {
    default_step->execute(rvholder<CT>
      (check,check));
  }
}
```

Listing 17

wisely, can help C++ code as well as good old style C. I personally have nothing against the wise use of macros, indeed they became very helpful while developing this framework.

Last but not least, the numeric value wrappers do not work with floating point numbers. This is due to the fact that extensive binary operations are used on the number to obfuscate its value and this would be impossible to accomplish with floating point values.

Some requirements

The code is written with 'older' compilers in mind, so not all the latest and greatest features of C++14 and 17 are included. CLang version 3.4.1 happily compiles the source code, so does g++ 4.8.2. Visual Studio 2015 is also compiling the code.

Unit testing is done using the Boost Unit test framework. The build system for the unit tests is CMake and there is support for code coverage (the last two were tested only under Linux).

License and getting the framework

The library is a header only library, released in the public domain under the MIT license. You can get it from https://github.com/fritzone/obfy

```
bool check_license1(const char* user,
                    const char* users_license)
{
  OBF_BEGIN
  std::string license;
  size_t ll = strlen(users_license);
  size_t l = strlen(user), lic_ctr = N(0);
  size_t add = N(0), i =N(0);

  FOR (V(i) = N(0), V(i) < V(ll), V(i)++)
    IF ( V(users_license[i]) != N(45) )
      license += users_license[i];
    ENDIF
  ENDFOR

  WHILE (V(lic_ctr) < license.length() )
    size_t i = lic_ctr;
    V(i) %= l;
    int current = 0;
    WHILE(V(i) < V(l) )
      V(current) += user[V(i)++];
    ENDWHILE
    V(current) += V(add);
    ++V(add);
    IF ( (license [lic_ctr]
        != letters[current % sizeof letters]) )
      RETURN(false);
    ENDIF
    lic_ctr++;
  ENDWHILE

  RETURN (true);
  OBF_END
}
```

Listing 18

Conclusion

History has shown us that if a piece of software is crackable, it will be cracked. And it just depends on the dedication, time spent, and effort invested by the software cracker when that piece of a software is to be proven crackable. There is no Swiss army knife when it comes to protecting your software against malicious interference because from the moment it left your build server and was downloaded, the software was out of your hands, and entered an uncontrollable environment. The only sensible thing you can do to protect your intellectual property is to make it as hard to crack as possible. This little framework provides a few ways of achieving this goal, and by making it open source, freely available and modifiable, to the developer community, we can only hope this will give it an advantage by allowing everyone to tailor it in order to suit their needs best. ■

Appendix: the license generating algorithm

As promised, Listing 19 is the naive license generating algorithm. Any further improvements to it are more than welcome.

References

[Andrivet] Random Generator by Sebastien Andrivet https://github.com/andrivet/ADVobfuscator

[Stackoverflow] http://stackoverflow.com/questions/12387239/reference-member-variables-as-class-members

[Wakely14] 'Stop the Constant Shouting' *Overload 121* June 2014, Jonathan Wakely

```
static const char letters[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
std::string generate_license(const char* user)
{
  if(!user) return "";
  // the license will contain only these character
  // 16 chars + 0
  char result[17] = { 0 };
  size_t l = strlen(user), lic_ctr = 0;
  int add = 0;
  while (lic_ctr < 16)
  {
    size_t i = lic_ctr;
    i %= l;
    int current = 0;
    while (i < l)
    {
      current += user[i];
      i++;
    }
    current += add;
    add++;
    result[lic_ctr] =
      letters[current % sizeof letters];
    lic_ctr++;
  }
  return std::string(result);
}
```

Listing 19

Afterwood

Comedy partnerships have a long history. Chris Oldwood considers their lessons for programmers.

When I think of some of the most memorable comedy acts, I instinctively go for the partnerships, such as Laurel & Hardy, Morecambe & Wise, The Two Ronnies, and The Chuckle Brothers. Okay, maybe that last suggestion isn't in my top 10 but they do spring to mind very quickly because they are another famous comedy partnership.

Does this mean that the best comedy only comes from partnerships? If I try and think of specific comedians then I'd possibly go with Bob Monkhouse, Steven Wright, Jimmy Carr or Jack Dee. Of course the face of the comedy act, the end product if you like, is the performance, the eventual delivery of the stream of gags from script to audience. What we might perceive as being a solo act, duo or group may just be the chosen form of delivery; behind the scenes the people that produce the actual content – the writers – may well be comprised of an entirely different number.

If you're bored enough to watch the credits at the end of a TV show, you'll often find there is more than one writer listed. Even if they acknowledge some writers under the separate heading of 'additional content', you're still likely to find the lion's share of the writing attributed to more than a single person. It's not uncommon for one half of a writing duo to be the most prominent face (e.g. Bob Monkhouse and Ricky Gervais), whereas the other half remains less well known because they are only supporting players in the performance (e.g. Denis Goodwin and Steven Merchant). Naturally, it doesn't stop at two, there are bigger teams of writers as well, but the point is that writing (and clearly not just in the field of comedy either) is commonly seen as a highly collaborative profession that benefits greatly from the input of many sources.

So why is writing software, which is also largely about communication, still often perceived as being a solo activity? Has the word of Eliyahu Goldratt [Goldratt84] (or more recently Kim, Behr & Spafford [Kim13]) about focusing on product flow instead of programmer utilisation still not reached in to the heart of the software industry's extensive management culture? Or is it ourselves, the legions of programmers, that are reluctant to give up our cubicles for fear of losing our identity?

In the past year I have done very little programming by myself. The vast majority of my time has been working in a pair, but I have also had the pleasure of doing a significant amount of mob programming too, usually in a group of four. I'm reaching a point now where the thought of having to work by myself makes me feel uncomfortable because I don't want to suffer the loss in productivity. It's still nice to do some background learning in the comfort of my own space but when it comes to delivering product features where the focus is on delivering working code to the customer, the joint effort is now feeling like a more natural way to go.

The reason it's taken so long (for me) to see the light has almost certainly been of my own making. Back when the ACCU conference was hosted in Oxford I remember a late night conversation in the bar (of course) where I posed the question about how the productivity of experienced programmers would benefit from practices like pair programming. The mistake I made back then was to think of two programmers as two CPUs sharing a problem – each additional CPU only adds another 60% (historically) due to communication overhead. But reading *The Goal* (and more recently *The Phoenix Project*) I realised my mistake was to think of myself as a resource to be utilised to 100% capacity, rather than leveraged to minimise the time to market of features (and therefore maximise the value extracted from each proposition).

Whilst I had always felt that being able to help unblock other people at the cost of not delivering as much personally was the right thing to do (a global optimisation) the management focus around individual performance always made it a choice which I was ill equipped to explain. Luckily books like Laurie William's *Pair Programming Illuminated* is becoming more well-known and has concrete data to back up the anecdotal evidence which has been floating around for much longer. This book, along with a number of other sources, came to my attention via talks given by Jon Jagger [Jagger16] and they have in turn been passed on to some of my clients that have also been sceptical of the practice. Their scepticism, like my own though, is usually borne out of looking for the answer to the wrong question.

As I suspect is the case with traditional writing partnerships, some work much better than others. Being a couple of decades into my professional programming career, I can't know how it would pan out for more junior developers but pairing and mobbing with experienced developers has mostly worked out extremely well. It's entirely possible that being mature freelancers, we're not worried about climbing the greasy pole and so we're entirely comfortable with just getting on with the task at hand and don't assume that any criticism is intended as a personal attack. We all have different backgrounds and that is something to be embraced, not diluted.

Programming in a group of two or more is definitely a skill in its own right. Just as with any conversation knowing when to speak and when to be silent is something you have to learn. Similarly if you currently have the keyboard you'll probably be bombarded with 'advice' and you'll need to learn to mediate. There are likely many things you'll want to pick up in the early days of your relationship, such as better ways to use the tooling and express concepts in code, and that's all on top of working together to solve the actual problem at hand, which should always be the primary focus. Personally I find the small scale scope creep trap all too easy to fall into and really appreciate having 'Gold Five' constantly reminding me in my ear to 'stay on target'.

One day I hope the software tooling world will catch up and each commit can read like the credits at the end of a TV show or film where all those who contributed to the feature are rightfully acknowledged, instead of just the one programmer who got to execute the final commit and push. And it's not just the programmers' names either; if you work in a cross-functional team you may well have a BA and QA providing valuable insights and direction which means your commit should be attributed to The Three Amigos.

The world of software is still dominated by the names of individuals, such as Linus Torvalds, Larry Wall and David Heinemeier Hansson. I wonder if in the future, when more of us start to work more closely with our fellow colleagues, we'll see a rise in the kind of partnerships that roll off the tongue, like Kernighan & Richie or Pike & Thompson? ■

Chris Oldwood Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

References

[Goldratt84] *The Goal* Eliyahu M. Goldratt (1984)

[Jagger16] http://jonjagger.blogspot.co.uk/2016/04/pair-programming-keynote.html

[Kim13] *The Phoenix Project* Gene Kim, Kevin Behr, George Spafford (2013)

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG